Micriµm

Empowering Embedded Systems

µC/FS

CPU independent
File System for embedded
applications

Software version 3.10

User's Manual Rev. 0

www.micrium.com

Disclaimer

Copyright notice

Trademarks

Registration

Please register the software via email. This way we can make sure you will receive updates or notifications of updates as soon as they become available.
For registration please provide the following information:

- Your full name and the name of your supervisor
- Your company name
- Your job title
- Your email address and telephone number
- Company name and address
- Your company's main phone number
- Your company's web site address
- Name and version of the product

Please send this information to: licensing@micrium.com

Contact address

Micrium
949 Crestview Circle
Weston, FL 33327-1848
U.S.A.
Phone : +1 954 217 2036
FAX : +1 954 217 2037
WEB : www.micrium.com
Email : support@micrium.com

Manual versions

If you find any errors in this document, please inform us and we will make the appropriate corrections for future releases.

For further information on topics or routines not yet specified, please contact us.

Print date: October 4, 2007

Manual version	Date	By	Explanation
3.10 R0	070927	SK	Chapter "API functions": * Storage layer functions added. Chapter "Running µC/FS on target hardware": * Structure/Directory names updated. Chapter "Device drivers": * Structure changed * Subsection "Resource usage" added to every driver section. * Section "NAND flash driver" updated and enhanced. * Section "NOR flash driver" updated and enhanced. * Section "Multimedia & SD card driver" enhanced. * Graphics updated. * Subsection Troubleshooting added. * Section "DataFlash driver" removed. The DataFlash driver is now integrated in the NAND driver. Chapter "Performance and resource usage": * Section "Memory footprint" updated.
3.08 R5	070719	SK	Chapter "Device drivers": * NAND: Pin description updated. * NAND: Illustrations added. * NOR: Illustrations added.
3.08 R4	070716	SK	Chapter "Introduction": * µC/FS structure picture changed. * Layer description updated.
3.08 R3	070703	SK	Chapter "API functions": * FS_InitStorage() updated. * FS_ReadSector() added. * FS_WriteSector() added. * FS_GetDeviceInfo() added. Chapter "Index" * Index updated.
3.08 R2	070703	SK	Chapter "Device driver": * "NAND flash driver" section enhanced.
3.08 R1	070618	SK	Chapter "API functions": * FS_UnmountLL added. * FS_GetVolumeStatus() added. * FS_InitStorage() added. Chapter "Porting µC/FS 2.x to 3.x" chapter.

Manual version	Date	By	Explanation
3.08 R0	070618	SK	Chapter "Introduction": * Section "Development environment" added. Chapter "API functions" updated. * FS_Mount() added. * FS_SetAutoMount() added. * FS_UnmountForced() added.
3.04 R0	070427	SK	Various improvements. Chapter "Running µC/FS on target hardware" updated. * Structural changes. * Section "Adjusting the RAM usage" added. Chapter "API functions" updated. * Samples updated. Chapter "Device driver" updated. * Generic flash driver renamed to NOR flash driver. - FS_FLASH_* replaced with FS_NOR_*. - NOR - additional driver functions added. * DataFlash driver added.
3.02 R0	070405	SK	Chapter "Running µC/FS on target hardware" updated. * Some smaller structural changes. * Section "Step 3: Add device driver" simplified. * Section "Step 4: Implement hardware routines" simplified. * Section "Troubleshooting" moved to chapter debugging. Chapter "API functions": * Section "File system configuration functions" added. - FS_AddDevice() moved into this section. - FS_AddPhysDevice() added. - FS_LOGVOL_Create() added. - FS_LOGVOL_AddDevice() added. Chapter "Device drivers": * Section "NAND": - FS_NAND_SetBlockRange() added. Chapter "Configuration of µC/FS": * Section "Compile-time configuration" - "Miscellaneous configuration" - "FS_NO_CLIB" default value corrected. Chapter "Debugging" - "FS_X_Log()", "FS_X_Warn()", "FS_X_ErrorOut()" : function description enhanced. Chapter "OS Support" updated.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler)
- The C programming language
- The target processor
- DOS command line.

If you feel that your knowledge of C is not sufficient, we recommend The C Programming Language by Kernighan and Richie (ISBN 0-13-1103628), which describes the standard in C-programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

This manual explains all the functions and macros that µC/FS offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command-prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
Reference	Reference to chapters, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections

Table 1.1: Typographic conventions

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction to µC/FS

1.1 What is µC/FS

µC/FS is a file system that can be used on any media for which you can provide basic hardware access functions.

µC/FS is a high-performance library that has been optimized for speed, versatility and memory footprint.

1.2 Features

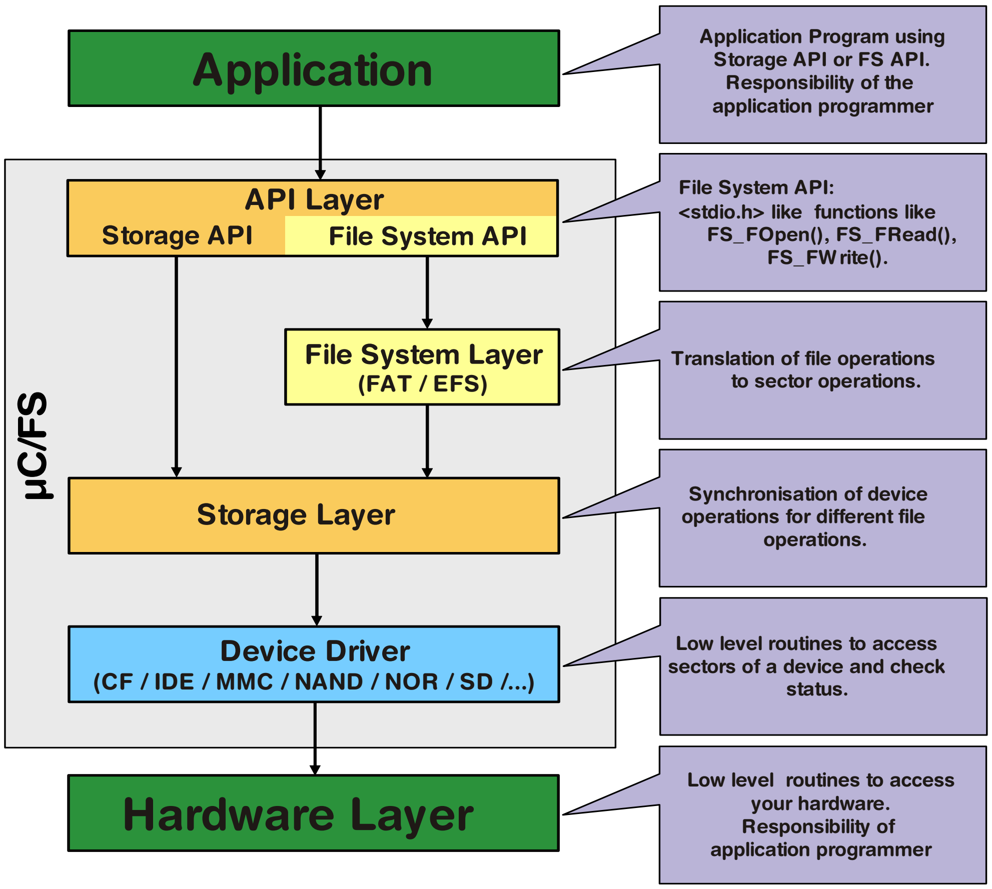
µC/FS is written in ANSI C and can be used on virtually any CPU.
Some features of µC/FS include:

- MS DOS/MS Windows-compatible FAT12, FAT16 and FAT32 support.
- An optional module that handles long file names of FAT media.
- Multiple device driver support. You can use different device drivers with µC/FS, which allows you to access different types of hardware with the file system at the same time.
- Multimedia support. A device driver allows you to access different media at the same time.
- OS support. µC/FS can be easily integrated into any OS. This allows using µC/FS in a multi-threaded environment.
- ANSI C stdio.h-like API for user applications. An application using the standard C I/O library can easily be ported to use µC/FS.
- Very simple device driver structure. µC/FS device drivers need only basic functions for reading and writing blocks. There is an empty generic example included. See `/device/generic/generic_drv.c` for more details.
- An optional generic device driver for SmartMedia cards or NAND flash devices, which can be easily used with any kind of (card reader) hardware.
- An optional generic device driver for Multimedia & SD cards using SPI mode that can be easily integrated.
- An optional generic IDE driver, which is also suitable for CompactFlash using either “True IDE” or “Memory Mapped” mode.
- An optional generic flash memory chip (EEPROM) driver that handles different flash sector sizes.
- An optional proprietary file system (EFS) with native long file name support.

1.3 Basic concepts

1.3.1 µC/FS structure

µC/FS is organized in different layers, illustrated in the diagram below. A short description of each layer's functionality follows below.


Application
Application Program using Storage API or FS API. Responsibility of the application programmer
µC/FS
API Layer
Storage API
File System API
File System API: <stdio.h> like functions like FS_FOpen(), FS_FRead(), FS_FWrite().
File System Layer (FAT / EFS)
Translation of file operations to sector operations.
Storage Layer
Synchronisation of device operations for different file operations.
Device Driver (CF / IDE / MMC / NAND / NOR / SD /...)
Low level routines to access sectors of a device and check status.
Hardware Layer
Low level routines to access your hardware. Responsibility of application programmer

API Layer

The API Layer is the interface between µC/FS and the user application. It is divided in two parts Storage API and File System API. The File System API declares file functions in ANSI C standard I/O style, such as `FS_FOpen()`, `FS_FWrite()` etc. The API Layer transfers any calls to these functions to the File System Layer. Currently the FAT file system or an optional file system, called EFS, are available for µC/FS. Right now they cannot be used simultaneously. The Storage API declares the functions which are required to initialize and access a storage medium. The Storage API allows sector read and write operations. The API Layer transfers these calls to the Storage Layer. The Storage API is optimized for applications which do not require file system functionality like file and directory handling. A typical application which uses the Storage API could be a USB mass storage device, where data has to be stored on a medium, but all file system functionality is handled by the host PC.

File System Layer

The file system layer translates file operations to logical block (sector) operations. After such a translation, the file system calls the logical block layer and specifies the corresponding device driver for a device.

Storage Layer

The main purpose of the Storage Layer is to synchronize accesses to a device driver. Furthermore, it provides a simple interface for the File System API. The Storage Layer calls a device driver to perform a block operation. It also contains the cache mechanism.

Device Driver

Device drivers are low-level routines that are used to access sectors of the device and to check status. It is hardware independent but depends on the storage medium.

Hardware Layer

These layer contains the low-level routines to access your hardware. These routines simply read and store fixed length sectors. The structure of the device driver is simple in order to allow easy integration of your own hardware.

1.3.2 Choice of file system type: FAT vs. EFS

Within µC/FS, there is a choice among two different file systems. The first, the FAT file system, is divided into three different sub types, FAT12, FAT16 and FAT32. While the other, EFS, is a proprietary file system developed by Micrium. Choosing a suitable file system will depend on the environment in which the end application is to operate.

The FAT file system was developed by Microsoft to manage file segments, locate available clusters and reassemble file for use. Released in 1976, the first version of the FAT file system was FAT12, which is no longer widely used. It was created for extremely small storage devices. (The early version of FAT12 did not support managing directories).

FAT16 is good for use on multiple operating systems because it is supported by all versions of Microsoft Windows, including DOS, OS/2 and Linux. The newest version, FAT32, improves upon the FAT16 file system by utilizing a partition/disk much more efficiently. It is supported by Microsoft Windows 98/ME/2000/XP and 2003 and as well on Linux based systems.

The EFS file system has been added to µC/FS as an alternative to the FAT file system. EFS has been designed for embedded devices. This file system reduces fragmentation of the data by utilizing drive space more efficiently, while still offering faster access to embedded storage devices. Another benefit of EFS is that there are no issues concerning long file name (LFN) support. The FAT file system was not designed for long file name support, limiting names to twelve characters (8.3). LFN support may be added to any of the FAT file systems, but there are legal issues that must be settled with Microsoft before end applications make use of this feature. Long file names are inherent to this proprietary file system relieving it of any legal issues.

1.3.3 Initializing the file system

The first thing that needs to be done after the system start-up and before any file-system function can be used, is to call the function `FS_Init()`. This routine initializes the internals of the filesystem. However it does not perform any low-level or device driver specific initialization.

1.3.4 Add your device

After initializing the file system, you have to add your target device to the file system. The function `FS_AddDevice()` adds and initializes the device.

1.4 Development environment (compiler)

The CPU used is of no importance; only an ANSI-compliant C compiler complying with at least one of the following international standard is required:

- ISO/IEC/ANSI 9899:1990 (C90) with support for C++ style comments (//)
- ISO/IEC 9899:1999 (C99)
- ISO/IEC 14882:1998 (C++)

If your compiler has some limitations, let us know and we will inform you if these will be a problem when compiling the software. Any compiler for 16/32/64-bit CPUs or DSPs that we know of can be used; most 8-bit compilers can be used as well.

A C++ compiler is not required, but can be used. The application program can therefore also be programmed in C++ if desired.

Chapter 2

Getting Started

This chapter provides a step-by-step introduction to using µC/FS.

2.1 Installation

µC/FS is shipped as a CD-ROM or as a `.zip` file in electronic form.
In order to install it, proceed as follows:

- If you received a CD, copy the entire contents to your hard drive into any folder of your choice. When copying, keep all files in their respective sub- directories. Make sure the files are not read-only after copying.
- If you received a .zip file, extract it to any folder of your choice, preserving the directory structure of the .zip file.

2.2 Using the Windows sample

If you have MS Visual C++ 6.00 or any later version available, you will be able to work with a Windows sample project using µC/FS. Even if you do not have the Microsoft compiler, you should read this chapter in order to understand how an application can use µC/FS.

2.2.1 Building the sample program

Open the workspace `FS_Start.dsw` with MS Visual Studio (for example double-clicking it). There is no further configuration necessary. You should be able to build the application without any error or warning message.

2.2.2 Stepping through the sample

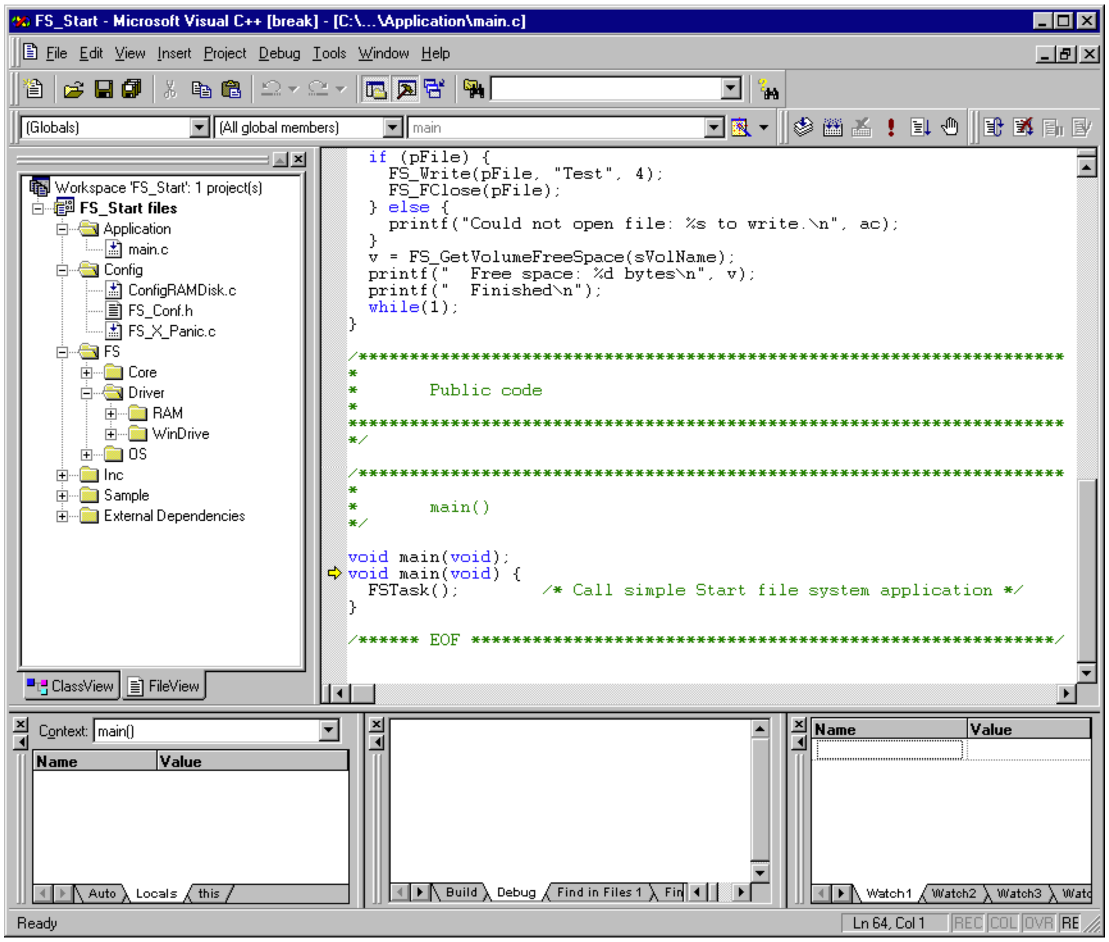
The sample project uses the RAM disk driver for demonstration. The `main` function of the sample application Start.c calls the function `FSTask()`. `FSTask()` initialises the file system and executes some basic file system operations.

`The sample application Start.c` step-by-step:

1. `FSTask() is called,`
2. `FSTask()` initialises and adds a device to µC/FS,
3. outputs the volume name,
4. calls FS_GetFreeVolumeSpace() and outputs the return value - the available free space of the RAM disk - to console window,
5. creates and opens a file test with write access (`File.txt`) on the device,
6. writes 4 bytes into the file and closes the file handle or outputs an error message,
7. calls FS_GetFreeVolumeSpace() and outputs the return value - the available free space of the RAM disk - again to console window,
8. outputs an quit message and runs into an endless loop

The sample step-by-step

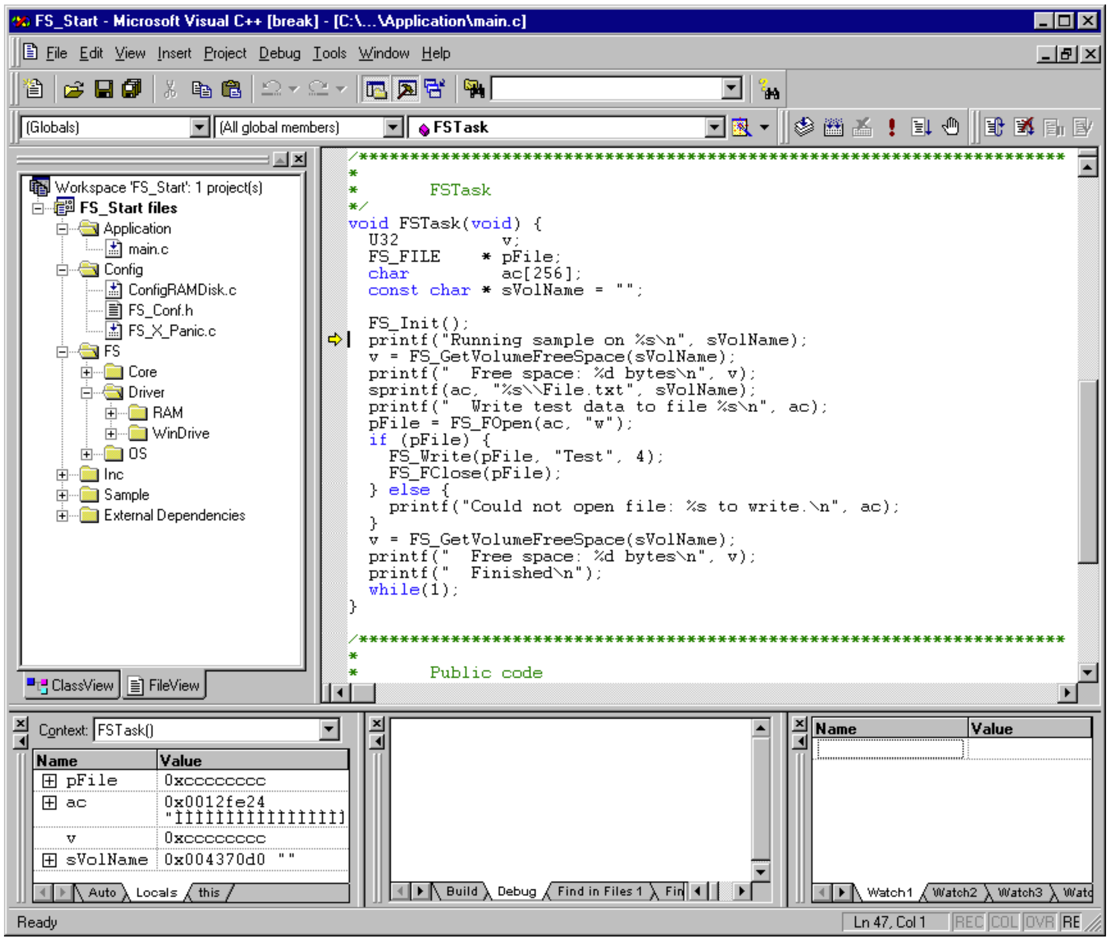
1. After starting the debugger by stepping into the application, your screen should look as the screenshot below. The `main` function calls `FSTask()`.



```
    if (pFile) {
      FS_Write(pFile, "Test", 4);
      FS_FClose(pFile);
    } else {
      printf("Could not open file: %s to write.\n", ac);
    }
    v = FS_GetVolumeFreeSpace(sVolName);
    printf("  Free space: %d bytes\n", v);
    printf("  Finished\n");
    while(1);
}

/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       main()
*/

void main(void);
void main(void) {
  FSTask();         /* Call simple Start file system application */
}

/****** EOF *********************************************************/
```

Figure 2.1: FS_Start project - main()

2. The first things called from `FSTask()` is the µC/FS function `FS_Init()`. This function initialises the file system and calls `FS_AddDevices()`. The function `FS_AddDevices()` is used to add and configure the used device drivers to the file system. In the example configuration only the RAM disk driver is added. `FS_Init()` must be called before using any other µC/FS function. You should step over this function.



```
/*********************************************************************
*
*       FSTask
*/
void FSTask(void) {
  U32          v;
  FS_FILE    * pFile;
  char         ac[256];
  const char * sVolName = "";

  FS_Init();
  printf("Running sample on %s\n", sVolName);
  v = FS_GetVolumeFreeSpace(sVolName);
  printf("  Free space: %d bytes\n", v);
  sprintf(ac, "%s\\File.txt", sVolName);
  printf("  Write test data to file %s\n", ac);
  pFile = FS_FOpen(ac, "w");
  if (pFile) {
    FS_Write(pFile, "Test", 4);
    FS_FClose(pFile);
  } else {
    printf("Could not open file: %s to write.\n", ac);
  }
  v = FS_GetVolumeFreeSpace(sVolName);
  printf("  Free space: %d bytes\n", v);
  printf("  Finished\n");
  while(1);
}

/*********************************************************************
*
*       Public code
```

Context: FSTask()

Name	Value
⊞ pFile	0xcccccccc
⊞ ac	0x0012fe24 "ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
v	0xcccccccc
⊞ sVolName	0x004370d0 ""

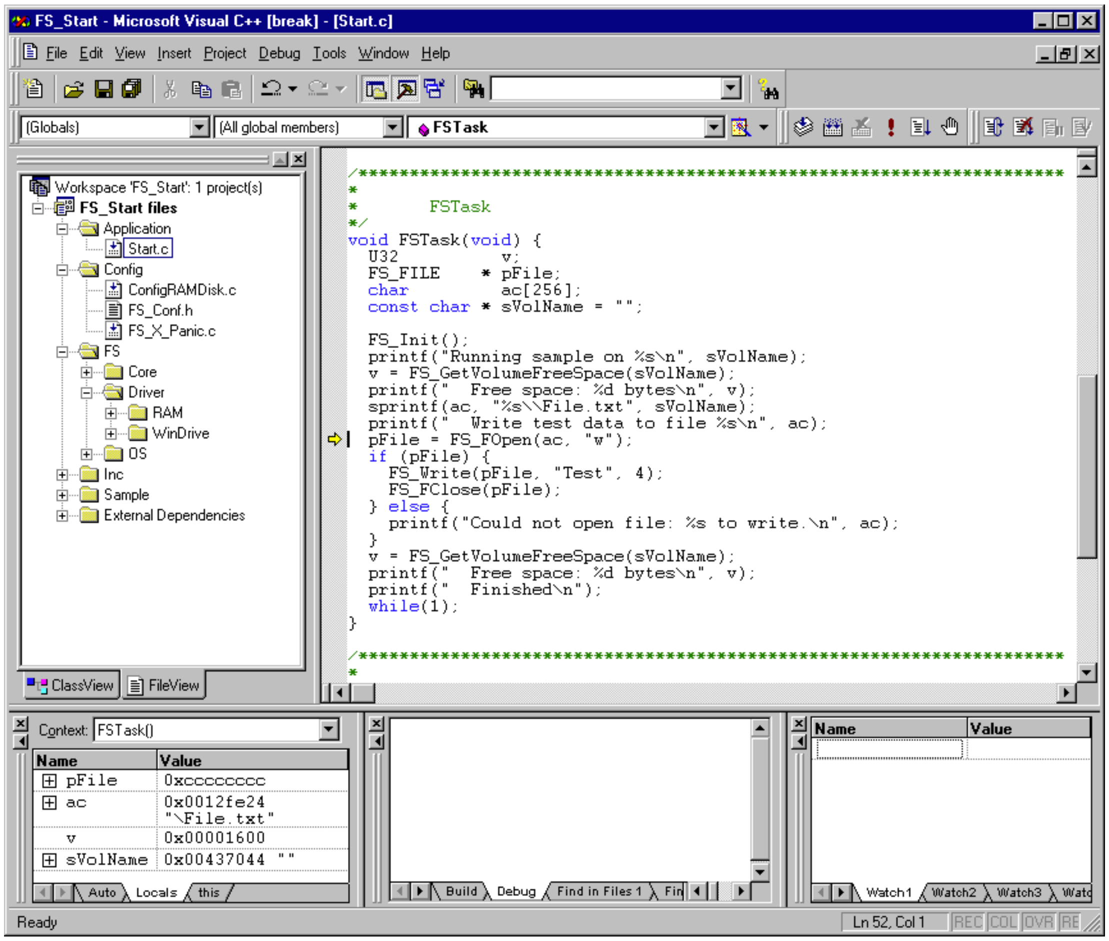
Auto Locals this

Name	Value

Figure 2.2: FS_Start project - FSTask()

3. If the initialisation was successfully, the volume name is printed in the console window.
4. The µC/FS function `FS_GetFreeSpace()` is called and the return value is written into the console window.

5. Afterwards, you should get to the µC/FS function call `FS_FOpen()`. This function creates a file named `file.txt` in the root directory of your RAM disk. Stepping over this function should return the address of an `FS_FILE` structure. In case of any error, it would return 0, indicating that the file could not be created.



FS_Start - Microsoft Visual C++ [break] - [Start.c]

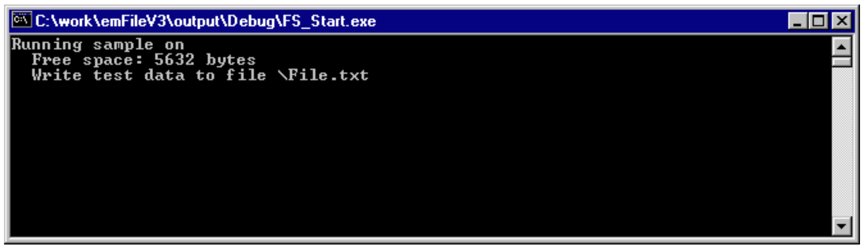
```
/*********************************************************************
*
*       FSTask
*/
void FSTask(void) {
  U32          v;
  FS_FILE    * pFile;
  char         ac[256];
  const char * sVolName = "";

  FS_Init();
  printf("Running sample on %s\n", sVolName);
  v = FS_GetVolumeFreeSpace(sVolName);
  printf("  Free space: %d bytes\n", v);
  sprintf(ac, "%s\\File.txt", sVolName);
  printf("  Write test data to file %s\n", ac);
  pFile = FS_FOpen(ac, "w");
  if (pFile) {
    FS_Write(pFile, "Test", 4);
    FS_FClose(pFile);
  } else {
    printf("Could not open file: %s to write.\n", ac);
  }
  v = FS_GetVolumeFreeSpace(sVolName);
  printf("  Free space: %d bytes\n", v);
  printf("  Finished\n");
  while(1);
}

/*********************************************************************
*
```

Context: FSTask()

Name	Value
pFile	0xcccccccc
ac	0x0012fe24 "\File.txt"
v	0x00001600
sVolName	0x00437044 ""

Figure 2.3: FS_Start project - FSTask()

6. If `FS_FOpen()` returns a valid pointer to an `FS_FILE` structure, the sample application will write a small ASCII string to this file by calling the µC/FS function `FS_FWrite()`. Step over this function. If a problem occurs, compare the return value of `FS_FWrite()` with the length of the ASCII string, which should be written. `FS_FWrite()` returns the number of elements which have been written.
   If no problem occurs the function µC/FS function `FS_FClose()` should be reached. `FS_FClose()` closes the file handle for `file.txt`. Step over this function.
7. Continue stepping over until you reach the call to the call of `FS_GetFreeSpace()`.The µC/FS function `FS_GetFreeSpace()` returns available free drive space in bytes. After you step over this function, the variable `v` should have a value greater than zero.
8. The return value is written in the console window.



C:\work\emFileV3\output\Debug\FS_Start.exe

```
Running sample on
  Free space: 5632 bytes
  Write test data to file \File.txt
```

Figure 2.4: FS_Start project - console output

Chapter 3

Running µC/FS on target hardware

This chapter explains how to integrate and run µC/FS on your target hardware. It explains this process step-by-step.

Integrating µC/FS

The µC/FS default configuration contains a single device: a RAM disk. This should always be the first step to check the proper function of µC/FS with your target hardware.

We assume that you are familiar with the tools you have selected for your project (compiler, project manager, linker, etc.). You should therefore be able to add files, add directories to the include search path, and so on. It is also assumed that you are familiar with the OS that you will be using in your target system (if you are using one). In this document the IAR Embedded Workbench® IDE is used for all examples and screenshots, but every other ANSI C toolchain can also be used. It is also possible to use make files; in this case, when we say "add to the project" , this translates into "add to the make file".

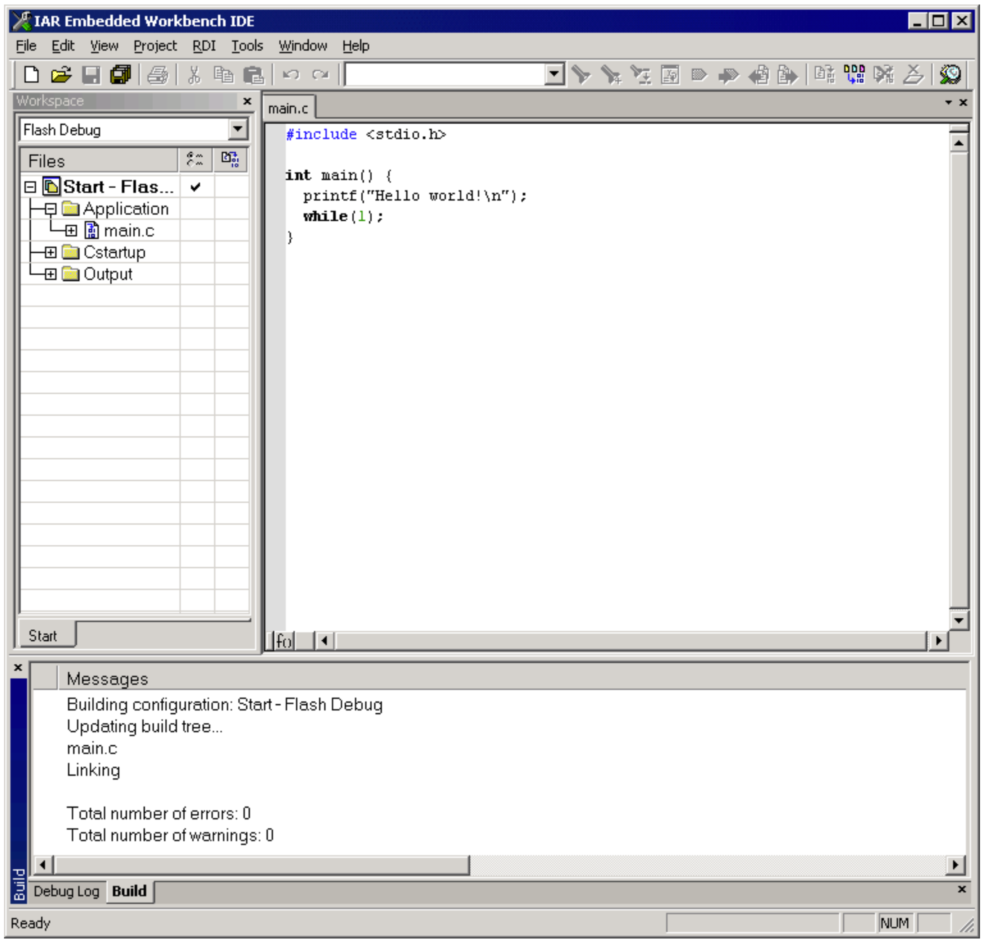
Procedure to follow

Integration of µC/FS is a relatively simple process, which consists of the following steps:

- Step 1: Creating a start project without µC/FS
- Step 2: Adding µC/FS to the start project
- Step 3: Adding the device driver
- Step 4: Activating the driver
- Step 5: Adjusting the RAM usage

3.1 Step 1: Creating a simple project without µC/FS

We recommend that you create a small "hello world" program for your system. That project should already use your OS and there should be a way to display text on a screen or serial port.



```
#include <stdio.h>

int main() {
  printf("Hello world!\n");
  while(1);
}
```

```
Messages
Building configuration: Start - Flash Debug
Updating build tree...
main.c
Linking

Total number of errors: 0
Total number of warnings: 0
```

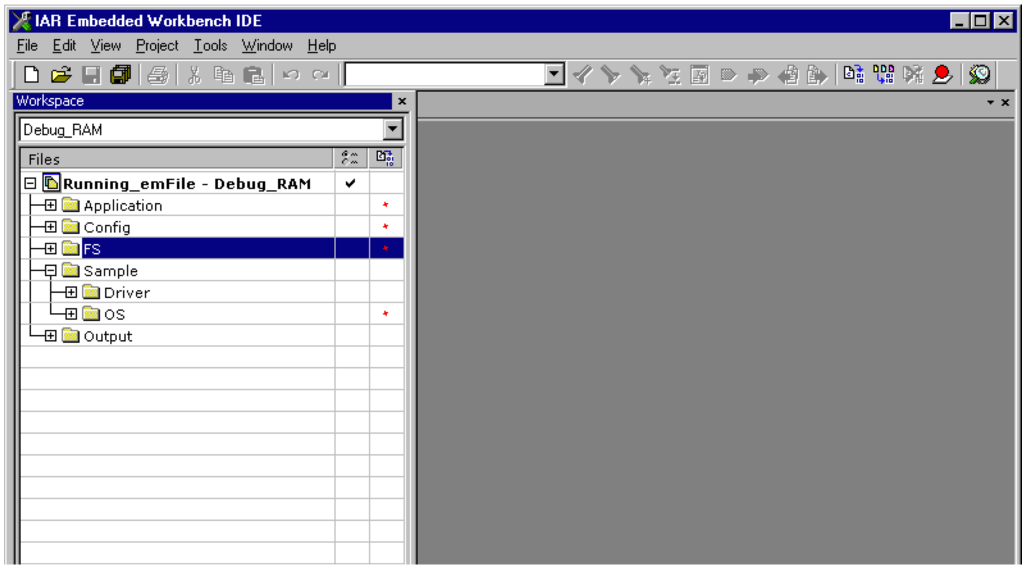
Figure 3.1: Start project

3.2 Step 2: Adding µC/FS to the start project

Add all source files in the following directories (and their subdirectories) to your project:

- `Application`
- `Config`
- `FS`
- `Sample\Driver\RAM`
- `Sample\OS\` (Optional, add if you use an RTOS. Add only the file compatible to the used operating system.)

It is recommended to keep the provided folder structure.


IAR Embedded Workbench IDE
File Edit View Project Tools Window Help
Workspace
Debug_RAM
Files

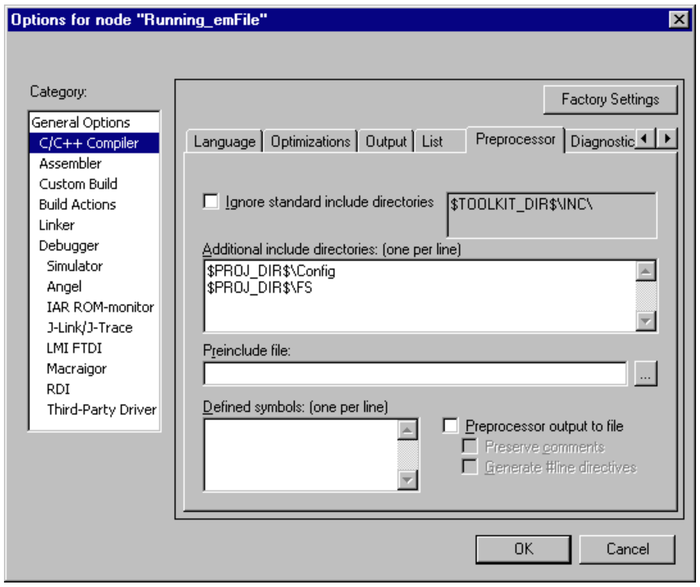
Running_emFile - Debug_RAM ✓
Application
Config
FS
Sample
Driver
OS
Output

Figure 3.2: µC/FS project structure

Configuring the include path

The include path is the path in which the compiler looks for include files. In cases where the included files (typically header files, `.h`) do not reside in the same directory as the C file to compile, an include path needs to be set. In order to build the project with all added files, you will need to add the following directories to your include path:

- `Config`
- `FS\`



Ignore standard include directories: [ ] — $TOOLKIT_DIR$\INC\

Additional include directories: (one per line)

```
$PROJ_DIR$\Config
$PROJ_DIR$\FS
```

Figure 3.3: Configure the include path

Select the start application

For quick and easy testing of your µC/FS integration, start with the code found in the folder `Application`. Exclude all files in the `Application` folder of your project except the supplied `main.c` and `Start`.c.

The application performs the following steps:

1. `main.c` calls `MainTask()`,
2. `MainTask()` initializes and adds a device to µC/FS,
3. checks if volume is low- level formatted and formats if required,
4. checks if volume is high-level formatted and formats if required,
5. outputs the volume name,
6. calls `FS_GetFreeVolumeSpace()` and outputs the return value - the available total space of the RAM disk - to console window,
7. creates and opens a file test with write access (`File.txt`) on the device,
8. writes 4 bytes into the file and closes the file handle or outputs an error message,
9. calls `FS_GetFreeVolumeSpace()` and outputs the return value - the available free space of the RAM disk - again to console window,
10. outputs an quit message and runs into an endless loop.

Build the project and test it

Build the project. It should compile without errors and warnings. If you encounter any problem during the build process, check your include path and your project configuration settings. The start application should print out the storage space of the device twice, once before a file has been written to the device and once afterwards.

3.3 Step 3: Adding the device driver

To configure µC/FS with a device driver 2 things need to be done at the same time:

- Adding device driver source to project
- Adding hardware routines to project

Every recommended step is explained in the following sections. For example, the implementation of the MMC/SD driver is shown, but all steps should be easy to adapt on every other device driver implementation.

3.3.1 Adding the device driver source to project

Add the driver sources to the project and add the directory to the include path.

Example

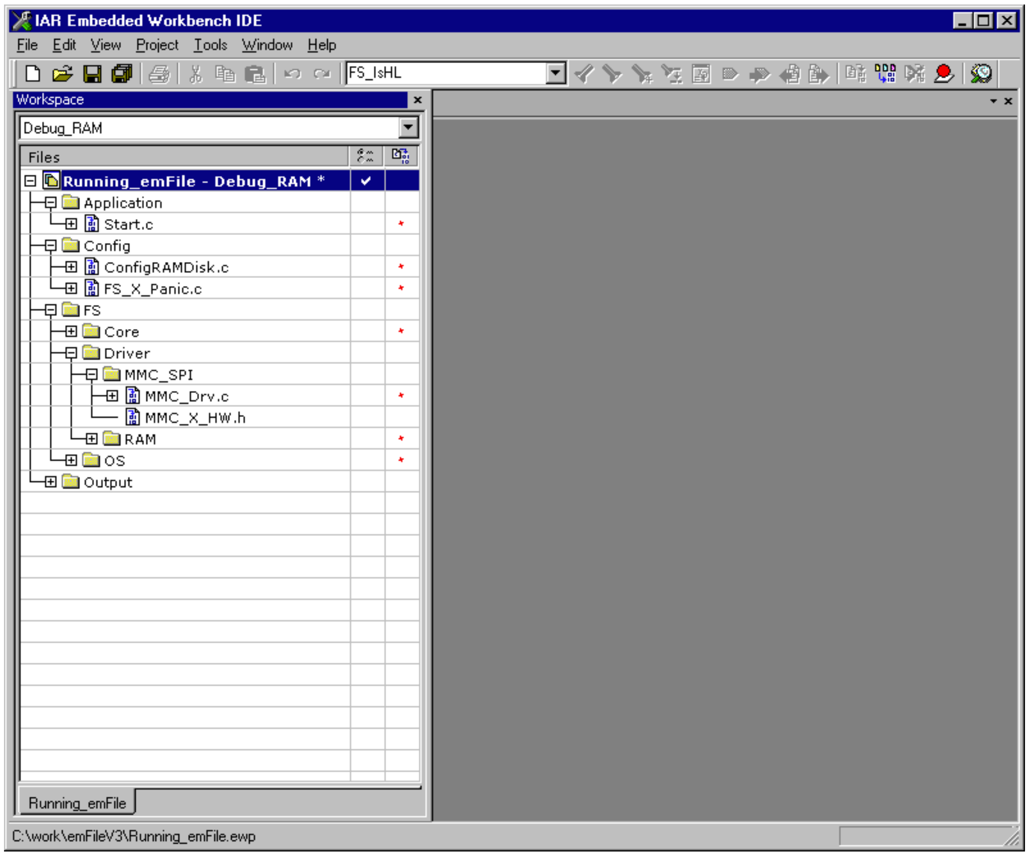


Figure 3.4: Add driver sources to project

Most drivers require additional hardware routines to work with the specific hardware. If your driver requires low-level I/O routines to access the hardware, you will have to provide them.

Drivers which require hardware routines are:

- NAND
- MMC/SD cards
- Compact flash / IDE

Drivers which not require hardware routines are:

- NOR flash
- RAM

Nearly all drivers have to be configured before they can be used. The runtime configuration functions which specify for example the memory addresses and the size of memory are located in the configuration file of the respective driver. All required configurations are explained in configuration section of the respective driver. If you use one of the drivers which do not require hardware routines skip the next section and refer to “Step 4: Activating the driver” on page 29.

3.3.2 Adding hardware routines to project

A template with empty function bodies and in most cases one ore more sample implementations are supplied for every driver which requires hardware routines. The easiest way to start is to use one of the ready-to-use samples. The ready-to-use samples can be found in the subfolders of `Sample\Driver\<DRIVER_DIR>\`. You should check the `Readme.txt` file located in the driver directory to see which samples are included. If there is one which is a good or close match for your hardware, it should be used. Otherwise, use the template to implement the hardware routines.

The template is a skeleton driver which contains empty implementations of the required functions and is the ideal base to start the implementation of hardware specific I/O routines.

What to do

Copy the compatible hardware function sample or the template into a subdirectory of your work directory and add it to your project. The template file is located in the `Sample\Driver\<DRIVER_DIR>\` directory; the example implementations are located in the respective directories. If you start the implementation of hardware routines with the hardware routine template, refer to “Device drivers” on page 141 for detailed information about the implementation of the driver specific hardware functions, else refer to section “Step 4: Activating the driver” on page 29.

Note: You cannot run and test the project with the new driver on your hardware as long as you have not added the proper configuration file for the driver to your project. Refer to section “Step 4: Activating the driver” on page 29 for more information about the activation of the driver with the configuration file.

3.4 Step 4: Activating the driver

After adding the driver source, and if required the hardware function implementation to the project, copy also the `Config<DRIVERNAME>.c` file (for example, `ConfigMMC_SPI.c` for the MMC/SD card driver using the SPI mode) into the `Config` directory of your µC/FS work directory. Add it afterwards to your project as show below.

Example

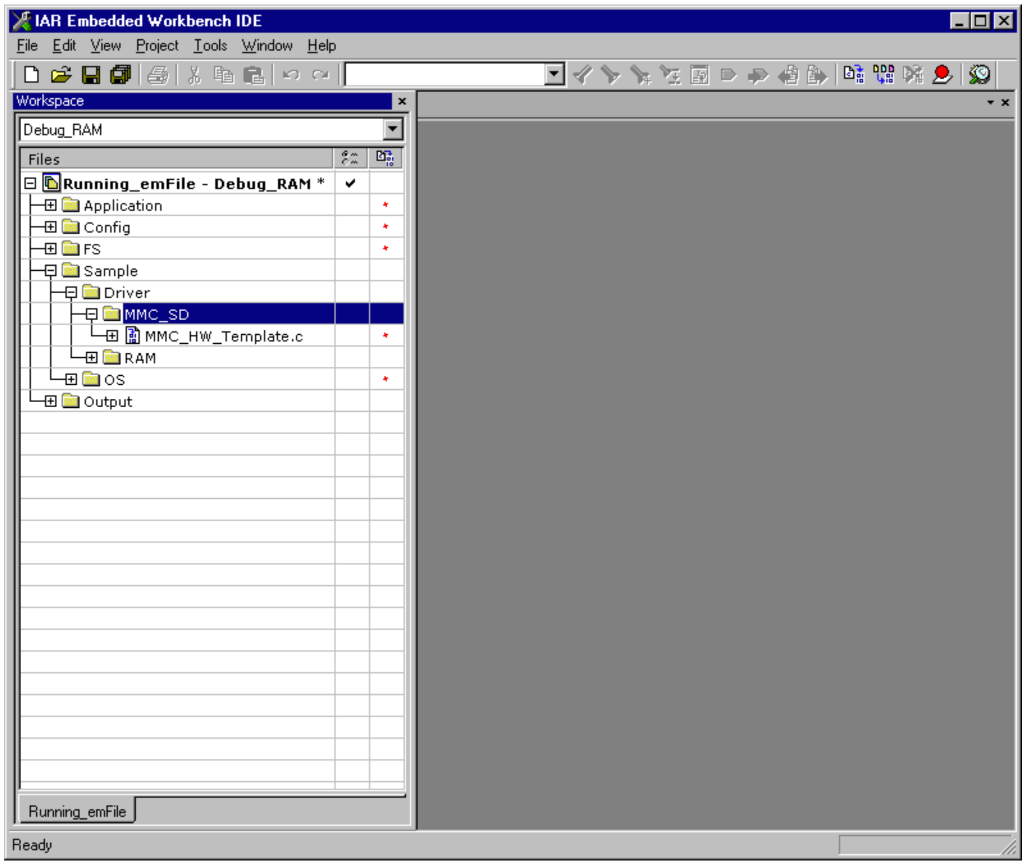


Figure 3.5: Adding template to your project

In the configuration files are all runtime configuration functions of the file system located. The configuration files include a start configuration which allows a quick and easy start with every driver. The most important function for the beginning is `FS_X_AddDevices()`. It activates and configures if required the driver. Driver which not not require hardware routines has to configured before they can be used.

3.4.1 Modifying the runtime configuration

The example on the next page adds a single CFI compliant NOR flash chip with a 16-bit interface and a size of 256 Mbytes to the file system. The base address, the start address and the size of the NOR flash are defined using the macros `FLASH0_BASE_ADDR`, `FLASH0_START_ADDR` and `FLASH0_SIZE`. Normally, only the `Defines, configurable` section of the configuration files requires changes for typical embedded systems. The `Public code` section which includes the time and date functions and `FS_X_AddDevices()` does not require modifications in most systems.

Example

```
/*********************************************************************
*
*       Defines, configurable
*
*       This section is the only section which requires changes for
*       typical embedded systems using the NOR flash driver with a
*       single device.
*
**********************************************************************
*/
#define ALLOC_SIZE         0x10000        // Size of memory dedicated to the file
                                          // system. This value should be fine-tuned
                                          // according for your system.
#define FLASH0_BASE_ADDR   0x40000000     // Base addr of the NOR flash device to
                                          // be used as storage
#define FLASH0_START_ADDR  0x40000000     // Start addr of the first sector be used
                                          // as storage. If the entire chip is
                                          // used for file system, it is identical to
                                          // the base addr.
#define FLASH0_SIZE        0x200000       // Number of bytes to be used for storage

/*********************************************************************
*
*       Public code
*
*       This section does not require modifications in most systems.
*
**********************************************************************
*/
/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*/
void FS_X_AddDevices(void) {
  //
  //  Add driver
  //
  FS_AddDevice(&FS_NOR_Driver);
  //
  //  Confgure the NOR flash interface
  //
  FS_NOR_SetPhyType(0, &FS_NOR_PHY_CFI_1x16);
  FS_NOR_Configure(0, FLASH0_BASE_ADDR, FLASH0_START_ADDR, FLASH0_SIZE);
}
```

After the driver has been added, the configuration functions (in this example `FS_NOR_SetPhyType()`and `FS_NOR_Configure()`) should be called. Detailed information about the driver configuration can be found in the configuration section of the respective driver.

Refer to section "Runtime configuration" on page 262 for detailed information about the other runtime configuration of the file system.

Before compiling and running the sample application with the added driver, you have to exclude `ConfigRAMDisk.c` from project.

Note for drivers which require hardware routines: If you have only added the template with empty function bodies until now, the project should compile without errors or warning messages. But you can only run the project on your hardware if you have finished the implementation of the hardware functions.

3.5 Step 5: Adjusting the RAM usage

The file system needs RAM for management purposes in various places. The amount of RAM required depends primarily on the configuration, especially the drivers used. The drivers which have their own level of management (such as NOR / NAND drivers) in general need more RAM than the "simple" drivers for hard drives, compact flash or SD cards.

Every driver needs to allocate RAM. The file system allocates RAM in the initialization phase and holds it while the file system is running. The macro `ALLOC_SIZE` which is located in the respective driver configuration file specifies the size of RAM used by the file system. This value should be fine-tuned according to the requirements of your target system.

What to do

Per default, `ALLOC_SIZE` is set to a value which should be appropriate for most target systems. Nevertheless, you should adjust it not to waste RAM. Once your file system project is up and running, you can check the real RAM requirement of the driver with the public auxiliary variable `FS_NumBytesAllocated` which is also located in the configuration file of the respective driver. Check the size of `FS_NumBytesAllocated` and adjust the value of `ALLOC_SIZE` to the value which `FS_NumBytesAllocated` has had after initialization of the file system.

Note: If you define `ALLOC_SIZE` with a value which is smaller than the appropriate size, the file system will run into `FS_X_Panic()`. If you define `ALLOC_SIZE` with a value which is above the limits of your target system, the linker will give an error during the build process of the project.

Chapter 4

API functions

In this chapter, you will find a description of each µC/FS API functions. An application should only access µC/FS by these functions.

4.1 API function overview

The table below lists the available API functions within their respective categories.

Function	Description
File system control functions	
`FS_Init()`	Starts the file system.
`FS_Mount()`	Mounts a volume.
`FS_SetAutoMount()`	Sets the mount behavior of the specified volume.
`FS_Unmount()`	Closes all file/directory handles and unmounts the volume.
`FS_UnmountForced()`	Invalidates all file/directory handles and unmounts the volume.
File system configuration functions	
`FS_AddDevice()`	Adds and makes a device driver accessable to µC/FS.
`FS_AddPhysDevice()`	Adds a device driver physical to µC/FS.
`FS_LOGVOL_Create()`	Creates a logical volume.
`FS_LOGVOL_AddDevice()`	Adds a device to a logical volume.
File access functions	
`FS_FClose()`	Closes a file.
`FS_FOpen()`	Opens a file.
`FS_FRead()`	Reads data from a file.
`FS_Read()`	Reads data from a file.
`FS_FWrite()`	Writes data to a file.
`FS_Write()`	Writes data to a file.
File positioning functions	
`FS_FSeek()`	Sets position of a file pointer.
`FS_FTell()`	Returns position of a file pointer.
`FS_GetFilePos()`	Returns position of a file pointer.
`FS_SetFilePos()`	Sets position of a file pointer.
Operations on files	
`FS_CopyFile()`	Copies a file.
`FS_GetFileAttributes()`	Retrieves the attributes of a given file/directory.
`FS_GetFileTime()`	Retrieves the creation, access or modify timestamp of a given file/directory.
`FS_GetFileTimeEx()`	Retrieves the timestamp of a given file/directory.
`FS_Move()`	Moves an existing file or a directory, including its children.
`FS_Remove()`	Deletes a file.
`FS_Rename()`	Renames a file/directory.
`FS_SetFileAttributes()`	Sets the attributes of a given file or directory.
`FS_SetFileTime()`	Sets the timestamp of a given file or directory.

Table 4.1: µC/FS API function overview

Function	Description
FS_SetFileTimeEx()	Sets the creation, access or modify timestamp of a given file or directory.
FS_SetEndOfFile()	Sets the end of a file.
FS_Truncate()	Truncates a file to a specified size.
FS_Verify()	Verifies a file with a given data buffer.
Directory functions	
FS_FindClose()	Closes a directory.
FS_FindFirstFile()	Searches for a file in a specified directory.
FS_FindNextFile()	Continues file search in a directory.
FS_MkDir()	Creates a directory.
FS_RmDir()	Removes a directory.
Formatting a medium	
FS_IsHLFormatted()	Checks if a device is high-level formatted.
FS_IsLLFormatted()	Checks if a device is low-level formatted.
FS_FormatLLIfRequired()	Checks if a device is low-level formatted and formats it if required.
FS_FormatLow()	Low-level formats a device.
FS_Format()	High-level formats a device.
File system extended functions	
FS_GetFileSize()	Retrieves the current file size of a given file pointer.
FS_GetNumVolumes()	Retrieves the available volumes.
FS_GetFreeSpace()	Gets amount of free space on drive.
FS_GetTotalSpace()	Gets total amount of drive space.
FS_GetVolumeFreeSpace()	Gets the free space of a given volume.
FS_GetVolumeInfo()	Get volume information (clusters, sectors etc.).
FS_GetVolumeLabel()	Retrieves the label of a given volume index.
FS_GetVolumeName()	Retrieves the name of a given volume index.
FS_GetVolumeSize()	Gets the size of a given volume.
FS_GetVolumeStatus	Returns the status of a volume.
FS_IsVolumeMounted()	Returns if the volume is mounted and has correct file system information.
FS_FileTimeToTimeStamp()	Converts a file time to a timestamp.
FS_SetBusyLEDCallback()	Sets a BusyLED callback for a specific volume.
FS_SetVolumeLabel()	Sets a label to a specific volume.
FS_TimeStampToFileTime()	Converts a timestamp to a file time.
Storage layer functions	
FS_STORAGE_GetDeviceInfo()	Returns the device info.
FS_STORAGE_Init()	Initializes the driver and OS if necessary.
FS_STORAGE_ReadSector()	Reads a sector from a device.
FS_STORAGE_ReadSectors()	Reads multiple sectors from a device.
FS_STORAGE_Sync()	Writes cached data to the storage medium.

Table 4.1: µC/FS API function overview(Continued)

Function	Description
FS_STORAGE_Unmount()	Low-level unmount. Unmounts a volume on driver layer.
FS_STORAGE_WriteSector()	Writes a sector from a device.
FS_STORAGE_WriteSectors()	Writes multiple sectors from a device.
FAT related functions	
FS_FAT_CheckDisk()	Checks and repairs a FAT volume.
FS_FAT_CheckDisk_ErrCode2Text()	Returns an error string to a specific Checkdisk error code.
FS_FAT_SupportLFN()	Add long file name support to the file system.
FS_FormatSD()	High-level formats a device according to the SD card file system specification.
File system cache functions	
FS_AssignCache()	Adds a cache to a specific device.
FS_CACHE_Clean()	Cleans the caches and writes dirty sectors to the volume.
FS_CACHE_SetMode()	Sets the mode for the cache.
FS_CACHE_SetQuota()	Sets the quotas for the different sector types in the `FS_Cache_RW_Quota` cache module.
Error-handling functions	
FS_ClearErr()	Clears the error status of a given file pointer.
FS_FEof()	Tests for end-of-file on a given file pointer.
FS_FError()	Returns the error code of a given file pointer.
FS_ErrorNo2Text()	Retrieves text for a given error code.
Obsolete functions	
FS_CloseDir()	Closes a directory stream.
FS_DirEnt2Attr()	Gets the directory entry attributes.
FS_DirEnt2Name()	Gets the directory entry name.
FS_DirEnt2Size()	Gets the directory entry file size.
FS_DirEnt2Time()	Gets the directory entry timestamp.
FS_GetDeviceInfo()	Returns the device info.
FS_GetNumFiles()	Gets the number of files in a directory.
FS_InitStorage()	Initializes the driver and OS if necessary.
FS_OpenDir()	Opens a directory stream.
FS_ReadDir()	Reads next directory entry.
FS_ReadSector()	Reads a sector from a device.
FS_RewindDir()	Resets position of directory stream.
FS_WriteSector()	Writes a sector to a device.
FS_UnmountLL()	Low-level unmount. Unmounts a volume on driver layer.

Table 4.1: µC/FS API function overview(Continued)

4.2 File system control functions

4.2.1 FS_Init()

Description

Starts the file system.

Prototype

```
void FS_Init (void);
```

Additional Information

`FS_Init()` initializes the file system and creates resources required for an OS integration of µC/FS. This function must be called before any other µC/FS function.

Example

```
#include "FS.h"

void main(void) {
  FS_Init();
  /*
     Access file system
  */
 }
```

4.2.2 FS_Mount()

Description

Mounts a volume.

Prototype

```
void FS_Mount (const char * sVolume);
```

Parameter	Description
`sVolume`	`sVolume` is the name of a volume. If not specified, the first device in the volume table will be used.

Table 4.2: FS_Mount() parameter list

Additional Information

This function can be useful if the default auto mount behavior has been changed with `FS_AutoMount()`. Normally, it is not required to mount a device with `FS_Mount()`, since the file system auto mounts all accessible volumes in read/write mode. Refer to "FS_SetAutoMount()" on page 39 for an overview about the different auto mount types.

4.2.3 FS_SetAutoMount()

Description

Sets the mount behavior of the specified volume.

Prototype

```
void FS_SetAutomount (const char * sVolume, U8 MountType);
```

Parameter	Description
sVolume	sVolume is the name of a volume. If not specified, the first device in the volume table will be used.
MountType	Specifies the auto mount behavior.

Table 4.3: FS_SetAutoMount() parameter list

Permitted values for parameter MountType	
FS_MOUNT_R	The volume will be read only auto mounted.
FS_MOUNT_RW	The volume will be read/write auto mounted.
0	Disables auto mount for the volume.

Additional Information

The file system auto mounts all volumes default in read/write mode.

4.2.4 FS_Unmount()

Description

Closes all file/directory handles and unmounts the volume.

Prototype

```
void FS_Unmount (const char * sVolume);
```

Parameter	Description
`sVolume`	`sVolume` is the name of a volume. If not specified, the first device in the volume table will be used.

Table 4.4: FS_Unmount() parameter list

Additional Information

`FS_Unmount()` should be called before a volume is removed. It guarantees that all file handles to this volume are closed and the directory entries for the files are updated. This function is also useful when shutting down the system.

Example

```
#include "FS.h"

void Shutdown(void) {
  FS_Unmount("");  /* Close all file handles and unmount the default volume. */
}
```

4.2.5 FS_UnmountForced()

Description

Invalidates all file/directory handles and unmounts the volume.

Prototype

```
void FS_UnmountForced (const char * sVolume);
```

Parameter	Description
sVolume	sVolume is the name of a volume. If not specified, the first device in the volume table will be used.

Table 4.5: FS_UnmountForced() parameter list

Additional Information

`FS_UnmountForced()` should be called if a volume has been removed before it could be regular unmounted. It invalidates all file handles. If you use `FS_UnmountForced()` there is no guarantee that all file handles to this volume are closed and the directory entries for the files are updated.

4.3 File system configuration functions

The file system control functions listed in this section can only be used in the runtime configuration phase. This means in practice that they can only be called from within `FS_X_AddDevices()`, refer to “FS_X_AddDevices()” on page 262 for more information about this function.

4.3.1 FS_AddDevice()

Description

Adds a device to µC/FS.

This consists of 2 operations:

1. Add physical device. This initialises the driver, allowing the driver to identify the storage device as far as required and allocate memory required for driver level management of the device. This makes sector operations possible.
2. Add the devices as a logical device. This makes it possible to mount the device, making it accessible for the file system and allowing file operations.

Prototype

```
FS_VOLUME * FS_AddDevice (const FS_DEVICE_TYPE * pDevType);
```

Parameter	Description
pDevType	Pointer to device driver table. See “Device driver function table” on page 258 for additional information.

Table 4.6: FS_AddDevice() parameter list

Return value

Pointer of the volume added to µC/FS.

Additional Information

This function can be used to add an additional device driver. You may also increase `FS_NUM_VOLUMES` to add additional space for more drives.

Example

```
#include "FS.h"

static int _DevGetName(U8 Unit) {
  return "";
}
static int _DevAddDevice(void) {
  if (_NumUnits >= NUM_UNITS) {
    return -1;
  }
  return _NumUnits++;
}

static int _DevRead(U8 Unit, U32 Sector, void *pBuffer) {
  return 0;
}

static int _DevWrite(U8 Unit, U32 Sector, void *pBuffer) {
  return 0;
}

static int _DevIoCtl(U8 Unit, I32 Cmd, I32 Aux, void *pBuffer) {
  return 0;
}
```

```
static int _DevIoCtl(U8 Unit, I32 Cmd, I32 Aux, void *pBuffer) {
  return 0;
}
static int _DevInitMedium(U8 Unit) {
  return 0;
}

static int _DevGetStatus(U8 Unit) {
  return 0;
}

static int _DevGetStatus(U8 Unit) {
  return 0;
}

struct FS_DEVICE_TYPE FS_xxx_Driver{
  const char *(*pfGetName)       (U8 Unit);
  int         (*pfAddDevice)     (void);
  int         (*pfRead)          (U8 Unit, U32 SectorNo, void *pBuffer,
                                  U32 NumSectors);
  int         (*pfWrite)         (U8 Unit, U32 SectorNo, const void *pBuffer,
                                  U32 NumSectors, U8 RepeatSame);
  int         (*pfIoCtl)         (U8 Unit, I32 Cmd, I32 Aux, void *pBuffer);
  int         (*pfInitMedium)    (U8 Unit);
  int         (*pfGetStatus)     (U8 Unit);
  int         (*pfGetNumUnits)   (void);
};

void AddDevices(void) {
  FS_AddDevice(&FS_xxx_Driver);
}
```

4.3.2 FS_AddPhysDevice()

Description

Adds a device physical to µC/FS. This initialises the driver, allowing the driver to identify the storage device as far as required and allocate memory required for driver level management of the device. This makes sector operations possible.

Prototype

```
int FS_AddPhysDevice (const FS_DEVICE_TYPE * pDevType);
```

Parameter	Description
pDevType	Pointer to device driver table. See "Device driver function table" on page 258 for additional information.

Table 4.7: FS_AddDevice() parameter list

Return value

>= 0: Unit number of the device.
<= 0: An error has occured.

Additional Information

Devices that are only physically added to µC/FS can be combined to a logical volume. Refer to "FS_LOGVOL_Create()" on page 45 and "FS_LOGVOL_AddDevice()" on page 46 for information about logical volumes.

4.3.3 FS_LOGVOL_Create()

Description

Creates a logical volume. A logical volume is the representation of one or more physical devices as a single device. It allows treating multiple physical devices as one larger device; the file system takes care of selecting the correct location on the correct physical device when reading or writing to the logical volume. Logical volumes are typically used if multiple flash devices (NOR or NAND) are present, but should be presented to the application the same way as single device with the combined capacity.

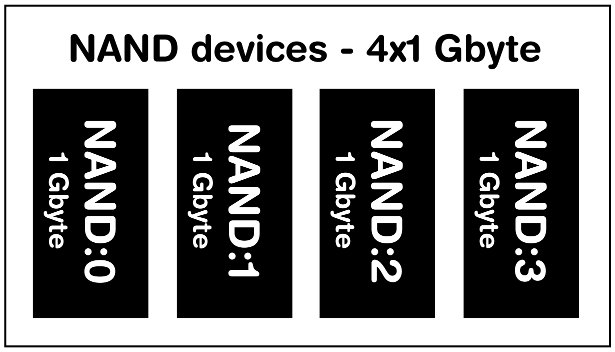
Prototype

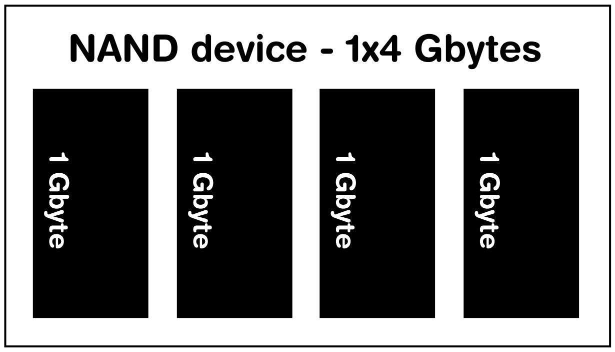
```
int FS_LOGVOL_Create (const char * sVolName);
```

Parameter	Description
`sVolName`	Name for the logical volume.

Table 4.8: FS_LOGVOL_Create() parameter list

Additional Information





Normally, all devices are added individually using `FS_AddDevice()`. This function adds the devices physically and logically to the file system, this means that every 1 Gbyte NAND devices can be accessed individually. Refer to “FS_AddDevice()” on page 42 for detailed information.

In contrast to adding all devices individually, all devices can be combined in a logical volume with a total size of all combined devices.

To create a logical volume the following steps have to be done:

1.The available device has to be physically added to the file system with `FS_AddPhysDevice()`.
2.A logical volume has to be created. with `FS_LOGVOL_Create()`.
3.The devices which are physically added to the file system has to be added to the logical volume with `FS_LOGVOL_AddDevice()`.

4.3.4 FS_LOGVOL_AddDevice()

Description

Adds a device to a logical volume.

Prototype

```
int FS_LOGVOL_AddDevice(const char *             sLogVolName,
                        const FS_DEVICE_TYPE * pDevice,
                        U8                       Unit,
                        U32                      StartOff,
                        U32                      NumSectors);
```

Parameter	Description
sVolName	Name of the logical volume.
pDevice	Pointer to device type that should be added.
Unit	Number of unit that should be added.
StartOff	Offset to define the start of sector range that should be used.
NumSector	Number of sectors that should be used.

Table 4.9: FS_LOGVOL_AddDevice() parameter list

Additional information

Only devices with an identical sector size can be combined to a logical volume. All additional added devices need to have the same sector size as the first physical device of the logical volume.

Example

```
void FS_X_AddDevices(void) {
  void  * pRAM;
  U8 Unit1, Unit2;
  //
  //  Add the RAM drives physical to FS
  //
  Unit1 = FS_AddPhysDevice(&FS_RAMDISK_Driver);
  Unit2 = FS_AddPhysDevice(&FS_RAMDISK_Driver);
  //
  //  Allocate the required memory and configure the RAM drives
  //
  pRAM = FS_Alloc(RAMDISK_NUM_SECTORS * RAMDISK_BYTES_PER_SECTOR);
  FS_RAMDISK_Configure(Unit1, pRAM, RAMDISK_BYTES_PER_SECTOR, RAMDISK_NUM_SECTORS);
  pRAM = FS_Alloc(RAMDISK_NUM_SECTORS * RAMDISK_BYTES_PER_SECTOR);
  FS_RAMDISK_Configure(Unit2, pRAM, RAMDISK_BYTES_PER_SECTOR, RAMDISK_NUM_SECTORS);
  //
  //  Create a logical volume to composite the RAM drives
  //
  FS_LOGVOL_Create("ramc");
  //
  //  Add the devices
  //
  FS_LOGVOL_AddDevice("ramc", &FS_RAMDISK_Driver, Unit1, 0, 0);
  FS_LOGVOL_AddDevice("ramc", &FS_RAMDISK_Driver, Unit2, 0, 0);

  if (FS_IsHLFormatted("ramc") == 0) {
    FS_Format("ramc", NULL);
  }
}
```

4.4 File access functions

4.4.1 FS_FClose()

Description

Closes an open file.

Prototype

```
int FS_FClose (FS_FILE * pFile);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE.

Table 4.10: FS_FClose() parameter list

Return value

== 0: File pointer has successfully been closed.
== 1: Failed to close the file pointer.

Example

```
void MainTask(void) {
  FS_FILE *pFile;

  pFile = FS_FOpen("test.txt", "r");
  if (pFile != 0) {
    /*
       access file
    */
    FS_FClose(pFile);'
  }
}
```

4.4.2 FS_FOpen()

Description

Opens an existing file or creates a new file depending on the parameters.

Prototype

```
FS_FILE *FS_FOpen (const char * pName,
                   const char * pMode);
```

Parameter	Description
pName	Pointer to a string that specifies the name of the file to create or open.
pMode	Mode for opening the file.

Table 4.11: FS_FOpen() parameter list

Return value

Returns the address of an `FS_FILE` data structure, if the file could be opened in the requested mode. `0` in case of any error.

Additional Information

A fully qualified file name looks like:
`[DevName:[UnitNum:]][DirPathList]Filename`

- `DevName` is the name of a device. If not specified, the first device in the volume table will be used.
  `UnitNum` is the number of the unit for this device. If not specified, unit 0 will be used. Note that it is not allowed to specify `UnitNum` if `DevName` has not been specified.
- `DirPathList` means a complete path to an already existing subdirectory; `FS_FOpen()` does not create directories. The path must start and end with a '`\`' character. Directory names in the path are separated by '`\`'. If `DirPathList` is not specified, the root directory on the device will be used.
- `FileName` desired
  If FAT is used and long file name support is not enabled, all file names and all directory names have to follow the standard FAT naming conventions (for example 8.3 notation).
  EFS supports long file names. The name length of a file or directory is limited to 235 valid characters.

The parameter `pMode` points to a string. If the string is one of the following, µC/FS will open the file in the specified mode:

Permitted values for parameter pMode	
r	Opens text file for reading.
w	Truncates to zero length or creates text file for writing.
a	Appends; opens/creates text file for writing at end-of-file.
rb	Opens binary file for reading.
wb	Truncates to zero length or creates binary file for writing.

Permitted values for parameter pMode	
ab	Appends; opens/creates binary file for writing at end-of-file.
r+	Opens text file for update (reading and writing).
w+	Truncates to zero length or creates text file for update.
a+	Appends; opens/creates text file for update, writing at end-of-file.
r+b or rb+	Opens binary file for update (reading and writing).
w+b or wb+	Truncates to zero length or creates binary file for update.
a+b or ab+	Appends; opens/creates binary file for update, writing at end-of-file.

For more details on the `FS_FOpen()` function, also refer to the ANSI C documentation regarding the `fopen()` function.
Note that µC/FS does not distinguish between binary and text mode; files are always accessed in binary mode.

Example

```
FS_FILE *pFile;

void foo1(void) {
  /* Open file for reading - default driver on default device */
  pFile = FS_FOpen("test.txt", "r");
}

void foo2(void) {
  /* Create new file for writing  - default driver on default device */
  pFile = FS_FOpen("test.txt", "w");
}

void foo3(void) {
  /* Open file for reading in folder 'mysub'
     - default driver on default device */
  pFile = FS_FOpen("\\mysub\\test.txt", "r");
}

void foo4(void) {
  /* Open file for reading - RAM device driver on default device */
  pFile = FS_FOpen("ram:test.txt", "r");
}

void foo5(void) {
  /* Open file for reading - RAM device driver on device number 2*/
  pFile = FS_FOpen("ram:1:test.txt", "r");
}
```

4.4.3 FS_FRead()

Description

Reads data from an open file.

Prototype

```
U32 FS_FRead (void *  pData,
              U32     Size,
              U32     N,
              U32 *   pFile);
```

Parameter	Description
pData	Pointer to a data buffer for storing data transferred from a file.
Size	Size of an element to be transferred from a file to a data buffer.
N	Number of elements to be transferred from the file.
pFile	Pointer to a data structure of type FS_FILE.

Table 4.12: FS_FRead() parameter list

Return value

Number of elements read.

Additional Information

If there is less data transferred than specified, you should check for possible errors by calling the FS_FError() function.

Example

```
char acBuffer[100];

void MainTask(void) {
  FS_FILE* pFile;
  int i;

  pFile = FS_FOpen("test.txt", "r");
  if (pFile != 0) {
    do {
      i = FS_FRead(acBuffer, 1, sizeof(acBuffer) - 1, pFile);
      acBuffer[i] = 0;
      if (i) {
        printf("%s", acBuffer);
      }
    } while (i);
    FS_FClose(pFile);
  }
}
```

4.4.4 FS_Read()

Description

Reads data from an open file.

Prototype

```
U32 FS_FRead (FS_FILE * pFile,
              void *    pData,
              U32       NumBytes);
```

Parameter	Description
pFile	Pointer to a data structure of type `FS_FILE`.
pData	Pointer to a data buffer for storing data transferred from a file.
NumBytes	Number of bytes to be transferred from the file.

Table 4.13: FS_Read() parameter list

Return value

Number of bytes read.

Additional Information

If there is less data transferred than specified, you should check for possible errors by calling the `FS_FError()` function.

Example

```
char acBuffer[100];

void MainTask(void) {
  FS_FILE *pFile;

  pFile = FS_FOpen("test.txt", "r");
  if (pFile != 0) {
    do {
      i = FS_Read(pFile, acBuffer, sizeof(acBuffer) - 1);
      acBuffer[i] = 0;
      if (i) {
        printf("%s", acBuffer);
      }
    } while (i);
    FS_FClose(pFile);
  }
}
```

4.4.5 FS_FWrite()

Description

Writes data to an open file.

Prototype

```
U32 FS_FWrite (const void *    pData,
               U32             Size,
               U32             N,
               FS_File *       pFile);
```

Parameter	Description
pData	Pointer to data to be written to the file.
Size	Size of an element to be transferred from a data buffer to a file.
N	Number of elements to be transferred to the file.
pFile	Pointer to a data structure of type FS_FILE.

Table 4.14: FS_FWrite() parameter list

Return value

Number of elements written.

Additional Information

If there is less data transferred than specified, you should check for possible errors by calling the `FS_FError()` function.

Example

```
const char acText[]="hello world\n";

void MainTask(void) {
  FS_FILE *pFile;

  pFile = FS_FOpen("test.txt", "w");
  if (pFile != 0) {
    FS_FWrite(acText, 1, strlen(acText), pFile);
    FS_FClose(pFile);
  }
}
```

4.4.6 FS_Write()

Description

Writes data to an open file.

Prototype

```
U32 FS_Write (FS_FILE *    pFile,
              const void * pData,
              U32          NumBytes);
```

Parameter	Description
`pFile`	Pointer to a data structure of type `FS_FILE`.
`pData`	Pointer to data to be written to the file.
`NumBytes`	Pointer to a data structure of type `FS_FILE`.

Table 4.15: FS_Write() parameter list

Return value

Number of bytes written.

Additional Information

If there is less data transferred than specified, you should check for possible errors by calling the `FS_FError()` function.

Example

```
const char acText[]="hello world\n";

void MainTask(void) {
  FS_FILE *pFile;

  pFile = FS_FOpen("test.txt", "w");
  if (pFile != 0) {
    FS_Write(pFile, acText, strlen(acText));
    FS_FClose(pFile);
  }
}
```

4.5 File positioning functions

4.5.1 FS_FSeek()

Description

Sets the current position of a file pointer.

Prototype

```
int FS_FSeek (FS_FILE * pFile,
              I32       Offset,
              int       Origin);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE.
Offset	Offset for setting the file pointer position.
Origin	Mode for positioning the file pointer.

Table 4.16: FS_FSeek() parameter list

Return value

== 0: If the file pointer has been positioned according to the parameters.
== -1: In case of any error.

Additional Information

The `FS_FSeek()` function moves the file pointer to a new location that is an offset in bytes from `Origin`. You can use `FS_FSeek()` to reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file.
Valid values for parameter `Origin` are:

Permitted values for parameter Origin	
FS_SEEK_SET	The origin is the beginning of the file.
FS_SEEK_CUR	The origin is the current position of the file pointer.
FS_SEEK_END	The origin is the current end-of-file position.

This function is identical to `FS_SetFilePos()`. Refer to “FS_SetFilePos()” on page 57 for more information.

Example

```
const char acText[]="some text will be overwritten\n";

void MainTask(void) {
  FS_FILE *pFile;

  pFile = FS_FOpen("test.txt", "w");
  if (pFile != 0) {
    FS_FWrite(acText, 1, strlen(acText), pFile);
    FS_FSeek(pFile, -4, FS_SEEK_CUR);
    FS_FWrite(acText, 1, strlen(acText), pFile);
    FS_FClose(pFile);
  }
}
```

4.5.2 FS_FTell()

Description

Returns the current position of a file pointer.

Prototype

```
I32 FS_FTell (FS_FILE * pFile);
```

Parameter	Description
pFile	Pointer to a data structure of type `FS_FILE`.

Table 4.17: FS_Tell() parameter list

Return value

>= 0: Current position of the file pointer in the file.
== -1: In case of any error.

Additional Information

In this version of µC/FS, this function simply returns the file pointer element of the file's `FS_FILE` structure. Nevertheless, you should not access the `FS_FILE` structure yourself, because that data structure may change in the future.
In conjunction with `FS_FSeek()`, this function can also be used to examine the file size. By setting the file pointer to the end of the file using `FS_SEEK_END`, the length of the file can now be retrieved by calling `FS_FTell()`.

This function is identical to `FS_GetFilePos()`. Refer to "FS_GetFilePos()" on page 56 for more information.

Example

```
const char acText[]="hello world\n";

void MainTask(void) {
  FS_FILE *pFile;
  I32 Pos;

  pFile = FS_FOpen("test.txt", "w");
  if (pFile != 0) {
    FS_FWrite(acText, 1, strlen(acText), pFile);
    Pos = FS_FTell(pFile);
    FS_FClose(pFile);
  }
}
```

4.5.3 FS_GetFilePos()

Description

Returns the current position of a file pointer.

Prototype

```
I32 FS_GetFilePos (FS_FILE * pFile);
```

Parameter	Description
`pFile`	Pointer to a data structure of type `FS_FILE`.

Table 4.18: FS_GetFilePos() parameter list

Return value

>= 0: Current position of the file pointer in the file.
== -1: In case of any error.

Additional Information

In this version of µC/FS, this function simply returns the file pointer element of the file's `FS_FILE` structure. Nevertheless you should not access the `FS_FILE` structure yourself, because that data structure may change in the future versions of µC/FS.
In conjunction with `FS_SetFilePos()`, `FS_GetFilePos()` this function can also be used to examine the file size. By setting the file pointer to the end of the file using `FS_SEEK_END`, the length of the file can now be retrieved by calling `FS_GetFilePos()`.

This function is identical to `FS_FTell()`. Refer to “FS_FTell()” on page 55 for more information.

Example

```
const char acText[]="hello world\n";

void MainTask(void) {
  FS_FILE *pFile;
  I32 Pos;

  pFile = FS_FOpen("test.txt", "w");
  if (pFile != 0) {
    FS_FWrite(acText, 1, strlen(acText), pFile);
    Pos = FS_GetFilePos(pFile);
    FS_FClose(pFile);
  }
}
```

4.5.4 FS_SetFilePos()

Description

Sets the current position of a file pointer.

Prototype

```
int FS_SetFilePos (FS_FILE * pFile,
                    I32       DistanceToMove,
                    int       MoveMethod);
```

Parameter	Description
pFile	Pointer to a data structure of type `FS_FILE`.
DistanceToMove	A 32-bit signed value where a positive value moves the file pointer forward in the file, and a negative value moves the file pointer backward.
MoveMethod	The starting point for the file pointer move.

Table 4.19: FS_SetFilePos() parameter list

Return value

== 0: If the file pointer has been positioned according to the parameters.
== -1: In case of any error.

Additional Information

The `FS_SetFilePos()` function moves the file pointer to a new location that is an offset in bytes from `MoveMethod`. You can use `FS_SetFilePos()` to reposition the pointer anywhere in a file. The pointer can also be positioned beyond the end of the file.
Valid values for parameter `MoveMethod` are:

Permitted values for parameter `MoveMethod`	
FS_FILE_BEGIN	The starting point is zero or the beginning of the file.
FS_FILE_CURRENT	The starting point is the current value of the file pointer.
FS_FILE_END	The starting point is the current end-of-file position.

This function is identical to `FS_FSeek()`. Refer to “FS_FSeek()” on page 54 for more information.

Example

```
const char acText[]="some text will be overwritten\n";

void MainTask(void) {
  FS_FILE *pFile;

  pFile = FS_FOpen("test.txt", "w");
  if (pFile != 0) {
    FS_FWrite(acText, 1, strlen(acText), pFile);
    FS_FSeek(pFile, -4, FS_SEEK_CUR);
    FS_FWrite(acText, 1, strlen(acText), pFile);
    FS_FClose(pFile);
  }
}
```

4.6 Operations on files

4.6.1 FS_CopyFile()

Description

Copies an existing file to a new file.

Prototype

```
int FS_CopyFile (const char * sSource,
                 const char * sDest);
```

Parameter	Description
sSource	Pointer to a string that specifies the name of an existing file.
sDest	Pointer to a string that specifies the name of the new file.

Table 4.20: FS_CopyFile() parameter list

Return value

== 0: If the file has been copied.
== -1: In case of any error.

Additional Information

Valid values for sSource are the same as for `FS_FOpen()`. Refer to "FS_FOpen()" on page 48 for examples of valid names.

Example

```
void MainTask(void) {
  FS_CopyFile("test.txt", "ram:\\info.txt");
}
```

4.6.2 FS_GetFileAttributes()

Description

The `FS_GetFileAttributes` function retrieves attributes for a specified file or directory.

Prototype

```
U8 FS_GetFileAttributes (const char * pName);
```

Parameter	Description
`pName`	Pointer to a string that specifies the name of a file or directory.

Table 4.21: FS_GetFileAttributes() parameter list

Return value

>= 0x00: Attributes of the given file or directory.
== 0xFF: In case of any error.
The attributes can be one or more of the following values:

Attribute	Description
`FS_ATTR_ARCHIVE`	The file or directory is an archive file or directory. Applications can use this attribute to mark files for backup or removal.
`FS_ATTR_DIRECTORY`	The given pName is a directory.
`FS_ATTR_HIDDEN`	The file or directory is hidden. It is not included in an ordinary directory listing.
`FS_ATTR_READ_ONLY`	The file or directory is read-only. Applications can read the file but cannot write to it or delete it. In case of a directory, applications cannot delete it
`FS_ATTR_SYSTEM`	The file or directory is part of, or is used exclusively by, the operating system.

Table 4.22: FS_GetFileAttributes() - list of possible attributes

Additional Information

Valid values for `pName` are the same as for `FS_FOpen()`. Refer to “FS_FOpen()” on page 48 for examples of valid names.

Example

```
void MainTask(void) {
  U8 Attributes;
  char ac[100];
  Attributes = FS_GetFileAttributes("test.txt");
  sprintf(ac, "File attribute of test.txt: %d", Attributes);
  FS_X_Log(ac);
}
```

4.6.3 FS_GetFileTime()

Description

Retrieves a timestamp for a specified file or directory.

Prototype

```
int FS_GetFileTime (const char * pName,
                    U32 *        pTimeStamp);
```

Parameter	Description
pName	Pointer to a string that specifies the name of a file or directory.
pTimeStamp	Pointer to a U32 variable that receives the timestamp.

Table 4.23: FS_GetFileTime() parameter list

Return value

== 0: The timestamp of the given file was successfully read and stored in pTimeStamp.
== -1: In case of any error.

Additional Information

Values for pName are the same as for FS_FOpen(). Refer to “FS_FOpen()” on page 48 for examples of valid names.

A timestamp is a packed value with the following format:

Bits	Description
0-4	Second divided by 2
5-10	Minute (0 - 59)
11-15	Hour (0-23)
16-20	Day of month (1-31)
21-24	Month (January -> 1, February -> 2, etc.)
25-31	Year offset from 1980. Add 1980 to get current year.

Table 4.24: FS_GetFileTime() - timestamp format description

To convert a timestamp to a “Structure FS_FILETIME” on page 99 structure, use the function “FS_TimeStampToFileTime()” on page 98.

Example

```
void MainTask(void) {
  char ac[80];
  U32 TimeStamp;
  FS_FILETIME FileTime;
  FS_GetFileTime ("test.txt", &TimeStamp);
  FS_TimeStampToFileTime(&TimeStamp, &FileTime);
  sprintf(ac, "File time of test.txt: %d-.2d-%.2d %.2d:%.2d:%.2d",
          FileTime.Year, FileTime.Month,  FileTime.Day,
          FileTime.Hour, FileTime.Minute, FileTime.Second);
  FS_X_Log(ac);
}
```

4.6.4 FS_GetFileTimeEx()

Description

Retrieves the creation, access or modify timestamp for a specified file or directory.

Prototype

```
int FS_GetFileTime (const char * pName,
                    U32 *         pTimeStamp
                    int           Index);
```

Parameter	Description
pName	Pointer to a string that specifies the name of a file or directory.
pTimeStamp	Pointer to a U32 variable that receives the timestamp.
Index	Flag that indicates which timestamp should be returned.

Table 4.25: FS_GetFileTimeEx() parameter list

Permitted values for parameter Index	
FS_FILETIME_CREATE	Indicates that the creation timestamp should be returned.
FS_FILETIME_ACCESS	Indicates that the access timestamp should be returned.
FS_FILETIME_MODIFY	Indicates that the modify timestamp should be returned.

Return value

== 0: The timestamp of the given file was successfully read and stored in pTimeS-tamp.
!= 0: In case of any error.

Additional Information

Values for pName are the same as for FS_FOpen(). Refer to "FS_FOpen()" on page 48 for examples of valid names.

A timestamp is a packed value with the following format:

Bits	Description
0-4	Second divided by 2
5-10	Minute (0 - 59)
11-15	Hour (0-23)
16-20	Day of month (1-31)
21-24	Month (January -> 1, February -> 2, etc.)
25-31	Year offset from 1980. Add 1980 to get current year.

Table 4.26: FS_GetFileTime() - timestamp format description

To convert a timestamp to a "Structure FS_FILETIME" on page 99 structure, use the function "FS_TimeStampToFileTime()" on page 98.

4.6.5 FS_Move()

Description

Moves an existing file or a directory, including its children.

Prototype

```
int FS_Move (const char * sExistingName,
             const char * sNewName);
```

Parameter	Description
sExistingname	Pointer to a string that names an existing file or directory.
sNewName	Pointer to a string that specifies the name of the new file or directory.

Table 4.27: FS_Move() parameter list

Return value

== 0: If the file was successfully moved.
== -1: In case of any error.

Additional Information

Valid values for `sExistingName` and `sNewName` are the same as for `FS_FOpen()`. Refer to "FS_FOpen()" on page 48 for examples of valid names. The `FS_Move()` function will move either a file or a directory (including its children) either in the same directory or across directories. The file or directory you want to move has to be on the same volume.

Example

```
void MainTask(void) {
  FS_Move("subdir1", "subdir2\\subdir3");
}
```

4.6.6 FS_Remove()

Description

Removes an existing file.

Prototype

```
int FS_Remove (const char * pName);
```

Parameter	Description
pName	Pointer to a string that specifies the file to be deleted.

Table 4.28: FS_Remove() parameter list

Return value

== 0: If the file was successfully removed.
== -1: In case of any error.

Additional Information

Valid values for pName are the same as for FS_FOpen(). Refer to "FS_FOpen()" on page 48 for examples of valid names.

Example

```
void MainTask(void) {
  FS_Remove("test.txt");
}
```

4.6.7 FS_Rename()

Description

Renames an existing file or a directory.

Prototype

```
int FS_Rename (const char * sExistingName,
               const char * sNewName);
```

Parameter	Description
sExistingName	Pointer to a string that names an existing file or directory.
sNewName	Pointer to a string that specifies the new name of the file or directory.

Table 4.29: FS_Rename() parameter list

Return value

== 0: If the file was successfully renamed.
== -1: In case of any error.

Additional Information

Valid values for `sExistingName` and `sNewName` are the same as for `FS_FOpen()`. Refer to "FS_FOpen()" on page 48 for examples of valid names. `sNewName` should only specify a valid file or directory name.

Example

```
void MainTask(void) {
  FS_Rename("ram:\\subdir1", "subdir2");
}
```

4.6.8 FS_SetFileAttributes()

Description

Sets attributes for a specified file or directory.

Prototype

```
int FS_SetFileAttributes (const char * pName,
                          U8           Attributes);
```

Parameter	Description
pName	Pointer to a string that specifies the name of a file or directory.
Attributes	Attributes to be set to the file or directory.

Table 4.30: FS_SetFileAttributes() parameter list

Permitted values for parameter Attributes	
FS_ATTR_ARCHIVE	The file or directory is an archive file or directory. Applications can use this attribute to mark files for backup or removal.
FS_ATTR_HIDDEN	The file or directory is hidden. It is not included in an ordinary directory listing.
FS_ATTR_READ_ONLY	The file or directory is read-only. Applications can read the file but cannot write to it or delete it. In case of a directory, applications cannot delete it.
FS_ATTR_SYSTEM	The file or directory is part of, or is used exclusively by, the operating system.

Return value

== 0: Attributes have been successfully set.
!= 0: In case of any error.

Additional Information

Valid values for pName are the same as for FS_FOpen(). Refer to "FS_FOpen()" on page 48 for examples of valid names.

Example

```
void MainTask(void) {
  U8 Attributes;
  char ac[100];
  FS_SetFileAttributes("test.txt", FS_ATTR_HIDDEN);
  Attributes = FS_GetFileAttributes("test.txt");
  sprintf(ac, "File attribute of test.txt: %d", Attributes);
  FS_X_Log(ac);
}
```

4.6.9 FS_SetFileTime()

Description

The `FS_SetFileTime` function sets the timestamp for a specified file or directory.

Prototype

```
int FS_SetFileTime (const char * pName,
                    U32          TimeStamp);
```

Parameter	Description
pName	Pointer to a string that specifies the name of a file or directory.
TimeStamp	Timestamp to be set to the file or directory.

Table 4.31: FS_SetFileTime() parameter list

Return value

== 0: The timestamp of the given file was successfully set.
!= 0: In case of any error.

Additional Information

Valid values for pName are the same as for `FS_FOpen()`. Refer to "FS_FOpen()" on page 48 for examples of valid names.
On a FAT medium, `FS_SetFileTime()` sets the creation time of a file or directory.
On a EFS medium, `FS_SetFileTime()` sets the time stamp of a file or directory.

A timestamp is a packed value with the following format.

Bits	Description
0-4	Second divided by 2
5-10	Minute (0 - 59)
11-15	Hour (0-23)
16-20	Day of month (1-31)
21-24	Month (January -> 1, February -> 2, etc.)
25-31	Year offset from 1980. Add 1980 to get current year.

Table 4.32: FS_SetFileTime() - timestamp format description

To convert a `FS_FILETIME` structure to a timestamp, use the function `FS_FileTimeToTimeStamp()`. For more information about the conversion have a look at the description of *"FS_FileTimeToTimeStamp()" on page 86*.

Example

```
void MainTask(void) {
  U32 TimeStamp;
  FS_FILETIME FileTime;

  FileTime.Year   = 2005;
  FileTime.Month  = 03;
  FileTime.Day    = 26;
  FileTime.Hour   = 10;
  FileTime.Minute = 56;
  FileTime.Second = 14;
  FS_FileTimeToTimeStamp (&FileTime, &TimeStamp);
  FS_SetFileTime("test.txt", TimeStamp);
}
```

4.6.10 FS_SetFileTimeEx()

Description

Sets the creation, access or modify timestamp for a specified file or directory.

Prototype

```
int FS_SetFileTimeEx (const char * pName,
                      U32 *         pTimeStamp
                      int           Index);
```

Parameter	Description
pName	Pointer to a string that specifies the name of a file or directory.
pTimeStamp	Pointer to a U32 variable that receives the timestamp.
Index	Flag that indicates which timestamp should be set.

Table 4.33: FS_SetFileTimeEx() parameter list

Permitted values for parameter Index	
FS_FILETIME_CREATE	Indicates that the creation timestamp should be set.
FS_FILETIME_ACCESS	Indicates that the access timestamp should be set.
FS_FILETIME_MODIFY	Indicates that the modify timestamp should be set.

Return value

== 0: The timestamp of the given file was successfully set.
!= 0: In case of any error.

Additional Information

Values for pName are the same as for FS_FOpen(). Refer to "FS_FOpen()" on page 48 for examples of valid names.

A timestamp is a packed value with the following format:

Bits	Description
0-4	Second divided by 2
5-10	Minute (0 - 59)
11-15	Hour (0-23)
16-20	Day of month (1-31)
21-24	Month (January -> 1, February -> 2, etc.)
25-31	Year offset from 1980. Add 1980 to get current year.

Table 4.34: FS_GetFileTime() - timestamp format description

To convert a timestamp to a FS_Filetime structure, use the function "FS_TimeStampToFileTime()" on page 98. For more information about the FS_FileTime structure, refer to "Structure FS_FILETIME" on page 99.

4.6.11 FS_SetEndOfFile()

Description

Sets the end of file for the specified file.

Prototype

```
int FS_SetEndOfFile (FS_FILE * pFile);
```

Parameter	Description
`pFile`	Pointer to a data structure of type FS_FILE.

Table 4.35: FS_SetEndOfFile() parameter list

Return value

== 0: End of File was set.
== -1: Operation failed.

Additional Information

`pFile` should point to a file that has been opened with write permission. Refer to “FS_FOpen()” on page 48. This function can be used to truncate or extend a file. If the file is extended, the contents of the file between the old EOF position and the new position are not defined.

Example

```
void MainTask(void) {
  FS_FILE * pFile;
  pFile = FS_FOpen("test.bin", "r+");
  FS_SetFilePos(pFile, 2000);
  FS_SetEndOfFile(pFile);
  FS_Fclose(pFile);
}
```

4.6.12 FS_Truncate()

Description

Truncates a file opened with `FS_FOpen()` to the specified size.

Prototype

```
int FS_Truncate (FS_FILE * pFile,
                 U32       NewSize);
```

Parameter	Description
`pFile`	Pointer to a data structure of type `FS_FILE`.
`NewSize`	New size of the file.

Table 4.36: FS_Truncate() parameter list

Return value

== 0: Truncation was successful.
== -1: Truncation failed.

Additional Information

This function truncates an open file. Be sure that `pFile` points to a file that has been opened with write permission. For more information about setting write permission for `pFile` have a look at the description of "FS_FOpen()" on page 48.

Example

```
void MainTask(void) {
  FS_FILE * pFile;
  U32    FileSize;
  Int       Success;
  pFile = FS_FOpen("test.bin", "r+");
  FileSize = FS_GetFileSize(pFile);
  Success = FS_Truncate(pFile, FileSize - 200);
  if (Success == 0) {
    FS_X_Log("Truncation was successful.");
  } else {
    FS_X_Log("Truncation was not successful");
  }
  FS_Fclose(pFile);
}
```

4.6.13 FS_Verify()

Description

Validates a file by comparing its contents with a data buffer.

Prototype

```
int FS_Verify (FS_FILE    * pFile,
               const void * pData,
               U32          NumBytes);
```

Parameter	Description
pFile	Pointer to a string that specifies the name of a file or directory.
pData	Pointer to a buffer that holds the data to be verified with the file.
NumBytes	Number of bytes to be verified.

Table 4.37: FS_Verify() parameter list

Return value

== 0: If verification was successful.
!= 0: Verification failed.

Additional Information

If the file contains less bytes than to be verified, only the available bytes are verified.

Note: The position of the file pointer has to set to the beginning of the data that should be verified.

Example

```
const U8 acVerifyData[4] = { 1, 2, 3, 4 };

void MainTask(void) {
  FS_FILE * pFile;
  I32         n;

  FS_Init();
  //
  // Open file and write data into
  //
  pFile = FS_FOpen("test.txt", "w+");
  FS_Write(pFile, acVerifyData, sizeof(acVerifyData));
  //
  // Determine current position of file pointer.
  //
  n = FS_FTell(pFile);
  //
  // Set file pointer to the start of the data that should be verified.
  //
  FS_FSeek(pFile, 0, FS_SEEK_SET);
  //
  // Verify data.
  //
  if (FS_Verify(pFile, acVerifyData, sizeof(acVerifyData)) == 0) {
    FS_X_Log("Verification was successful");
  } else {
    FS_X_Log("Verification failed");
  }
  //
  // Set file pointer to end of data that was written and verified.
  //
  FS_FSeek(pFile, n, FS_SEEK_SET);
  FS_FClose(pFile);

  while (1);
}
```

4.7 Directory functions

4.7.1 FS_FindClose()

Description

Closes a directory.

Prototype

```
void FS_FindClose (FS_FIND_DATA * pfd);
```

Parameter	Description
pfd	Pointer to a FS_FIND_DATA structure.

Table 4.38: FS_FindClose() parameter list

Example

```
typedef struct {
  // Public elements, to be used by application
  U8     Attributes;
  U32    CreationTime;
  U32    LastAccessTime;
  U32    LastWriteTime;
  U32    FileSize;
  char * sFileName;
  // Private elements. Not be used by the application
  int SizeofFileName;
  FS__DIR Dir;
} FS_FIND_DATA;

FindFileSample(void) {
  FS_FIND_DATA fd;
  char acFilename[20];

  if (FS_FindFirstFile(&fd, "\\YourDir\\", acFilename, sizeof(acFilename)) == 0) {
    do {
      printf(acFilename);
    } while (FS_FindNextFile (&fd));
  }
  FS_FindClose(&fd);
}
```

4.7.2 FS_FindFirstFile()

Description

Searches files in a specified directory.

Prototype

```
int FS_FindFirstFile (FS_FIND_DATA * pfd,
                      const char *   sPath,
                      char *         sFilename,
                      int            sizeofFilename);
```

Parameter	Description
pfd	Pointer to a `FS_FIND_DATA` structure.
sPath	Pointer to a string containing the name of a directory which should be scanned.
sFilename	Pointer to a buffer used to store the name of a file which has been found.
sizeofFilename	Size of the buffer which contains the name of a file which has been found.

Table 4.39: FS_FindFirstFile() parameter list

Return value

== 0: The directory or file found.
!= 0: In case of any error.

Additional Information

A fully qualified directory name looks like:

`[DevName:[UnitNum:]][DirPathList]DirectoryName`

where:

- `DevName` is the name of a device, for example `"ram"` or `"mmc"`. If not specified, the first device in the device table will be used.
  `UnitNum` is the number for the unit of the device. If not specified, unit 0 will be used. Note that it is not allowed to specify `UnitNum` if `DevName` has not been specified.
- `DirPathList` is a complete path to an existing subdirectory. The path must start and end with a '\' character. Directory names in the path are separated by '\'. If `DirPathList` is not specified, the root directory on the device will be used.
- `DirectoryName` and all other directory names have to follow the standard FAT naming conventions (for example 8.3 notation), if support for long file names is not enabled.

To open the root directory, simply use an empty string for `sPath`.

Refer to "Structure FS_FIND_DATA" on page 77 for more information about the structure pfd points to.

Example

Refer to "FS_FindClose()" on page 72 for an example.

4.7.3 FS_FindNextFile()

Description

Continues a file search from a previous call to the `FindFirstFile()` function.

Prototype

```
int FS_FindNextFile (FS_FIND_DATA * pfd);
```

Parameter	Description
`pfd`	Pointer to a `FS_FIND_DATA` structure.

Table 4.40: FS_FindNextFile() parameter list

Return value

== 1: File found in directory.
== 0: In case of any error.

Example

Refer to "FS_FindClose()" on page 72 for an example.

4.7.4 FS_MkDir()

Description

Creates a new directory.

Prototype

```
int FS_MkDir (const char * pDirName);
```

Parameter	Description
pDirName	Fully qualified directory name.

Table 4.41: FS_MkDir() parameter list

Return value

== 0: The directory was successfully created.
== -1: In case of any error.

Additional Information

Refer to "FS_FindFirstFile()" on page 73 for examples of valid fully qualified directory names. Note that FS_MkDir() will not create the whole pDirName, it will only create a directory in an already existing path.

Example

```
void FSTask1(void) {
  int Err;
  /* Create mydir in directory test - default driver on default device */
  Err = FS_MkDir("\\test\\mydir");
}

void FSTask2(void) {
  int Err;
  /* Create directory mydir - RAM device driver on default device */
  Err = FS_MkDir("ram:\\mydir");
}
```

4.7.5 FS_RmDir()

Description

Deletes a directory.

Prototype

```
int FS_RmDir (const char * pDirname);
```

Parameter	Description
pDirname	Fully qualified directory name.

Table 4.42: FS_RmDir() parameter list

Return value

== 0: If the directory has been successfully removed.
== -1: In case of any error.

Additional Information

Refer to "FS_FindFirstFile()" on page 73 for examples of valid and fully qualified directory names. `FS_RmDir()` will only delete a directory if it is empty.

Example

```
void FSTask1(void) {
  int Err;
  /* Remove mydir in directory test - default driver on default device */
  Err = FS_RmDir("\\test\\mydir");
}

void FSTask2(void) {
  int Err;
  /* Remove directory mydir - RAM device driver on default device */
  Err = FS_RmDir("ram:\\mydir");
}
```

4.7.6 Structure FS_FIND_DATA

Description

The FS_FORMAT_INFO structure represents the information used to access directories and files.

Prototype

```
typedef struct {
  // Public elements, to be used by application
  U8      Attributes;
  U32     CreationTime;
  U32     LastAccessTime;
  U32     LastWriteTime;
  U32     FileSize;
  char *  sFileName;
  // Private elements. Not be used by the application
  int SizeofFileName;
  FS__DIR Dir;
} FS_FIND_DATA;
```

Members	Description
Attributes	Specifies the file attributes of the file found.
CreationTime	U32 value containing the time the file was created.
LastAccessTime	U32 value containing the time that the file was last accessed.
LastWriteTime	U32 value containing the time that the file was last written to.
FileSize	U32 value specifies the size of the file.
sFileName	String that is the name of the file.
SizeofFileName	Size of the file name. (Private element. Not to be used by application.)
Dir	Directory administration structure. (Private element. Not to be used by an application.)

Table 4.43: FS_FIND_DATA - list of structure elements

4.8 Formatting a medium

In general, before a medium can be used to read or write to a file, it needs to be formatted. Flash cards are usually already preformatted and do not need to be formatted. Flashes used as storage devices have normally to be reformatted. These devices require a low-level format first, then a high-level format. The low-level format is device-specific, the high-level format depends on the file system only. (FAT-format typically).

4.8.1 FS_IsHLFormatted()

Description

Checks if the volume is high-level formatted.

Prototype

```
int FS_IsHLFormatted (const char * sVolume);
```

Parameter	Description
pVolume	Name of the device to check.

Table 4.44: FS_IsHLFormatted() parameter list

Return value

== 1: Volume is high-level formatted.
== 0: Volume is not high-level formatted.
==-1: Device is not ready or a general error has occured.

4.8.2 FS_IsLLFormatted()

Description

Checks if the volume is low-level formatted.

Prototype

```
int FS_IsHLFormatted (const char * sVolume);
```

Parameter	Description
sVolume	Name of the device to check.

Table 4.45: FS_IsLLFormatted() parameter list

Return value

== 1: Volume is low-level formatted.
== 0: Volume is not low-level formatted.
== -1: Volume does not require a low-level format or in case of any error.

Additional Information

Low-level format is only required for devices which have their own management level. These are the drivers for NOR flashes, NAND flashes. MMC, SD and all other cards do not require a low-level format.

4.8.3 FS_FormatLLIfRequired()

Description

Checks if the volume is low-level formatted and formats the volume if required.

Prototype

```
int FS_FormatLLIfRequired (const char * pVolumeName);
```

Parameter	Description
pVolumeName	Name of the device to low-level format.

Table 4.46: FS_FormatLLIfRequired() parameter list

Return value

== 0: Ok - low-level format successful.
== 1: Low-level format not required.
== -1: Volume does not require a low-level format or in case of any error.

Additional Information

Low-level format is only required for devices which have their own management level. These are the drivers for NOR flashes, NAND flashes. MMC, SD and all other cards do not require a low-level format.

4.8.4 FS_FormatLow()

Description

Low-level formats a device. Required by NAND/NOR flashes prior to format.

Prototype

```
int FS_FormatLow (const char * pDeviceName);
```

Parameter	Description
pVolumeName	Name of the device to low-level format.

Table 4.47: FS_FormatLow() parameter list

Return value

== 0: Low-level format successful.
!= 0: In case of any error.

Additional Information

Low-level format is only required for devices which have their own management level. These are the drivers for NOR flashes, NAND flashes and SMC cards. MMC, SD and all other cards do not require a low-level format.

4.8.5 FS_Format()

Description

Performs a high-level format of a device. This means putting the management information required by the File system on the medium. In case of FAT, this means primarily initialization of FAT and the root directory, as well as the BIOS parameter block.

Prototype

```
int FS_Format (const char *     pDeviceName
               FS_FORMAT_INFO * pFormatInfo);
```

Parameter	Description
pVolumeName	Name of the device to format.
pFormatInfo	Optional info for formatting.

Table 4.48: FS_Format() parameter list

Return value

== 0: High-level format successful.
!= 0: In case of any error.

Additional Information

There are many different ways to format a medium, even with one file system. If the second parameter is not specified, reasonable default values are used (auto-format). However, `FS_Format()` also allows fine-tuning of the parameters used. For details, refer to the sample file `Format.c`, which is shipped with µC/FS.

For more information about the structure `FS_FORMAT_INFO`, refer to “Structure FS_FORMAT_INFO” on page 84.

4.8.6 Structure FS_FORMAT_INFO

Description

The `FS_FORMAT_INFO` structure represents the information used to format a volume.

Prototype

```
typedef struct {
  U16 SectorsPerCluster;
  U16 NumRootDirEntries;
  FS_DEV_INFO * pDevInfo;
} FS_FORMAT_INFO;
```

Members	Description
`SectorsPerCluster`	A cluster is the minimal unit size a file system can handle. Sectors are combined together to form a cluster.
`NumRootDirEntries`	Represents the number of directory entries the root directory should have. Typically it is only used for FAT12 and FAT16 drives. FAT32 has a dynamically grow table.
`pDevInfo`	Pointer to a `FS_DEV_INFO` structure.

Table 4.49: FS_FORMAT_INFO - list of structure elements

4.8.7 Structure FS_DEV_INFO

Description

The `FS_DEV_INFO` structure contains the medium information.

Prototype

```
typedef struct {
  U16 NumHeads;
  U16 SectorsPerTrack;
  U32 NumSectors;
  U16 BytesPerSector;
} FS_DEV_INFO;
```

Members	Description
`NumHeads`	Number of heads on the drive. This is relevant for mechanical drives only.
`SectorsPerTrack`	Number of sectors in each track. This is relevant for mechanical drives only.
`NumSectors`	Total number of sectors on the medium.
`BytesPerSector`	Number of bytes per sector.

Table 4.50: FS_DEV_INFO - list of structure elements

4.9 Extended functions

4.9.1 FS_FileTimeToTimeStamp()

Description

Converts a given `FS_FILE_TIME` structure to a timestamp.

Prototype

```
void FS_FileTimeToTimeStamp(const FS_FILETIME * pFileTime,
                            U32 *               pTimeStamp);
```

Parameter	Description
`pFileTime`	Pointer to a data structure of type `FS_FILETIME`, that holds the data to be converted.
`pTimeStamp`	Pointer to a `U32` variable to store the timestamp.

Table 4.51: FS_FileTimeToTimeStamp() parameter list

Additional Information

Refer to “Structure FS_FILETIME” on page 99 to get information about the `FS_FILETIME` data structure.

4.9.2 FS_GetFileSize()

Description

Gets the current file size of a file.

Prototype

```
U32 FS_GetFileSize (FS_FILE * pFile);
```

Parameter	Description
pFile	Pointer to a data structure of type FS_FILE.

Table 4.52: FS_GetFileSize() parameter list

Return Value

>= 0: File size in bytes (0 - 0xFFFFFFFE).
== 0xFFFFFFFF: In case of any error.

4.9.3 FS_GetFreeSpace()

Description

Gets the amount of free space on a specific device.

Prototype

```
U32 FS_GetFreeSpace (const char * pDevice);
```

Parameter	Description
`pDevice`	Name of a device.

Table 4.53: FS_GetFreeSpace() parameter list

Return Value

Amount of free space in bytes. Free space larger than 4 Gbytes is truncated to 0xFFFFFFFF (the maximum value of a `U32`).

4.9.4 FS_GetNumVolumes()

Description

Retrieves the number of available volumes.

Prototype

```
int FS_GetNumVolumes (void);
```

Return Value

The number of available volumes.

Additional Information

This function can be used to get the name of each available volume. Refer to "FS_GetVolumeName()" on page 94 for getting more information.

4.9.4.1 FS_GetTotalSpace()

Description

Gets the amount of total space on a specific device.

Prototype

```
U32 FS_GetTotalSpace (const char * pDevice);
```

Parameter	Description
`pDevice`	Name of a device.

Table 4.54: FS_GetTotalSpace() parameter list

Return Value

Number of total space in bytes. If the total space is larger than 4 Gbytes, it is truncated to 0xFFFFFFFF (the maximum value of an `U32`).

4.9.5 FS_GetVolumeFreeSpace()

Description

Gets amount of free space on a specific volume.

Prototype

```
U32 FS_GetVolumeFreeSpace (const char * sVolume);
```

Parameter	Description
`sVolume`	Pointer to a string that specifies the volume name.

Table 4.55: FS_GetVolumeFreeSpace() parameter list

Return Value

> 0: Amount of free space in bytes. Free space larger than 4 GB is reported as 0xFFFFFFFF (the maximum value of a `U32`).
== 0: If the volume cannot be found.

Additional Information

Note that free space larger than four Gbytes is reported as 0xFFFFFFFF because a `U32` cannot represent bigger values. The function `FS_GetVolumeInfo()` can be used for larger spaces. If you do not need to know if there is more than 4 GB of free space available, you can still reliably use `FS_GetVolumeFreeSpace()`.

Valid values for `sVolume` have the following structure:

```
[DevName:[UnitNum:]]
```

where:

- `DevName` is the name of a device. If not specified, the first device in the volume table will be used.
- `UnitNum` is the number of the unit of the device. If not specified, unit 0 will be used.

Note that to specify `UnitNum`, also `DevName` has to be specified.

4.9.6 FS_GetVolumeInfo()

Description

Gets volume information, that is the number of clusters (total and free), sectors per cluster, and bytes per sector. The function collects volume information and stores it into the given `FS_DISK_INFO` structure.

Prototype

```
int FS_GetVolumeInfo (const char *   sVolume,
                      FS_DISK_INFO * pInfo);
```

Parameter	Description
`sVolume`	Volume name as a string.
`pInfo`	Pointer to a `FS_DISK_INFO` structure.

Table 4.56: FS_GetVolumeInfo() parameter list

Return Value

== 0: If volume information could be retrieved successfully.
== -1: In case of error, for example if the volume could not be found.

Additional Information

Valid values for `sVolume` have the following structure:

`[DevName:[UnitNum:]]`

where:

- `DevName` is the name of a device. If not specified, the first device in the volume table will be used.
- `UnitNum` is the number of the unit of the device. If not specified, unit 0 will be used.

Note that to specify `UnitNum`, also `DevName` has to be specified.

Example

```
#include "FS.h"
#include <stdio.h>

void MainTask(void) {
  FS_DISK_INFO Info;

  if (FS_GetVolumeInfo("ram:", &Info) == -1) {
    printf("Failed to get volume information.\n");
  } else {
    printf("Number of total clusters = %d\n"
           "Number of free clusters  = %d\n"
           "Sectors per cluster      = %d\n"
           "Bytes per sector         = %d\n",
           Info.NumTotalClusters,
           Info.NumFreeClusters,
           Info.SectorsPerCluster,
           Info.BytesPerSector);
    }
}
```

4.9.7 FS_GetVolumeLabel()

Description

Returns a volume label name if one exists.

Prototype

```
int FS_GetVolumeLabel (const char *   sVolume,
                       char *         pVolumeLabel
                       unsigned       VolumeLabelSize);
```

Parameter	Description
sVolume	Volume name as a string.
pVolumeLabel	Pointer to a buffer to receive the volume label.
pVolumeLabelSize	Size of the buffer which can used to store pVolumeLabel.

Table 4.57: FS_GetVolumeLabel() parameter list

Return Value

> 0: Number of total bytes available on this volume.
== -1: Failed to get volume information.

4.9.8 FS_GetVolumeName()

Description

Retrieves the name of the specified volume index.

Prototype

```
int FS_GetVolumeName (int    Index,
                      char * pBuffer,
                      int    BufferSize);
```

Parameter	Description
Index	Index number of the volume.
pBuffer	Pointer to a buffer that receives the null-terminated string for the volume name.
BufferSize	Size of the buffer to receive the null terminated string for the volume name.

Table 4.58: FS_GetVolumeName() parameter list

Return Value

If the function succeeds, the return value is the length of the string copied to pBuffer, excluding the terminating null character, in bytes.
If the pBuffer buffer is too small to contain the volume name, the return value is the size of the buffer required to hold the volume name plus the terminating null character. Therefore, if the return value is greater than BufferSize, make sure to call the function again with a buffer that is large enough to hold the volume name.

Example

```
void ShowAvailableVolumes(void) {
  int NumVolumes;
  int i;
  int BufferSize;
  char acVolume[12];

  BufferSize = sizeof(acVolume);
  NumVolumes = FS_GetNumVolumes();
  FS_X_Log("Available volumes:\n");
  for (i = 0; i < NumVolumes; i++) {
    if (FS_GetVolumeName(i, &acVolume[0], BufferSize) < BufferSize) {
      FS_X_Log(acVolume);
      FS_X_Log("\n");
    }
  }
}
```

4.9.9 FS_GetVolumeSize()

Description

Gets the total size of a specific volume.

Prototype

```
U32 FS_GetVolumeSize (const char * sVolume);
```

Parameter	Description
`sVolume`	Volume name as a string.

Table 4.59: FS_GetVolumeSize() parameter list

Return Value

Volume size in bytes. Volume sizes larger than 4 Gbyte are truncated to 0xFFFFFFFF (the maximum value of a `U32`).

Additional Information

Note that volume sizes larger than 4 Gbytes are reported as 0xFFFFFFFF because a U32 cannot represent bigger values. The function `FS_GetVolumeInfo()` can be used for larger media. If you do not need to know if the total space is bigger than 4 Gbytes, you can still reliably use `FS_GetVolumeSize()`.

Valid values for `sVolume` have the following structure:

```
[DevName:[UnitNum:]]
```

where:

- `DevName` is the name of a device. If not specified, the first device in the volume table will be used.
- `UnitNum` is the number of the unit of the device. If not specified, unit 0 will be used.

Note that to specify `UnitNum`, also `DevName` has to be specified.

4.9.10 FS_GetVolumeStatus()

Description

Returns the status of a volume.

Prototype

```
int FS_GetVolumeStatus(const char  * sVolume);
```

Parameter	Description
`sVolume`	Volume name as a string.

Table 4.60: FS_GetVolumeStatus() parameter list

Return Value

Return value	Description
`FS_MEDIA_STATE_UNKNOWN`	The volume state is unknown.
`FS_MEDIA_NOT_PRESENT`	A volume is not present.
`FS_MEDIA_IS_PRESENT`	A volume is present.

Table 4.61: FS_GetVolumeStatus() - list of return values

4.9.11 FS_IsVolumeMounted()

Description

Returns if a volume was succesfully mounted and has correct file system information.

Prototype

```
int FS_IsVolumeMounted (const char * sVolumeName);
```

Parameter	Description
sVolumeName	Volume name as a string.

Table 4.62: FS_IsVolumeMounted() parameter list

Return Value

== 1: If volume information is mounted.
== 0: In case of error, for example if the volume could not be found, is not detected, or has incorrect file system information.

Example

```
#include "FS.h"
#include <stdio.h>

void MainTask(void) {
  if (FS_IsVolumeMounted("ram:")) {
    printf("Volume is already mounted.\n");
 } else {
    printf("Volume is not mounted.\n");
 }
}
```

4.9.12 FS_TimeStampToFileTime()

Description

Converts a given timestamp to a `FS_FILE_TIME` structure.

Prototype

```
void FS_TimeStampToFileTime (U32           TimeStamp,
                             FS_FILETIME * pFileTime);
```

Parameter	Description
`TimeStamp`	Timestamp to be converted.
`pFileTime`	Pointer to a data structure of type `FS_FILETIME` to store the converted timestamp.

Table 4.63: FS_TimeStampToFileTime() parameter list

Additional Information

A `TimeStamp` is a packed value with the following format:

Bits	Description
`0-4`	Second divided by 2
`5-10`	Minute (0 - 59)
`11-15`	Hour (0 - 23)
`16-20`	Day of month (1 - 31)
`21-24`	Month (January -> 1, February -> 2, etc.)
`25-31`	Year offset from 1980. Add 1980 to get current year.

Table 4.64: FS_TimeStampToFileTime() - timestamp format description

4.9.13 Structure FS_FILETIME

Description

The `FS_FILETIME` structure represents a timestamp using individual members for the month, day, year, weekday, hour, minute, and second. This can be useful for getting or setting a timestamp of a file or directory.

Prototype

```
typedef struct {
  U16 Year;
  U16 Month;
  U16 Day;
  U16 Hour;
  U16 Minute;
  U16 Second;
} FS_FILETIME;
```

Members	Description
`Year`	Represents the year. The year must be greater than 1980.
`Month`	Represents the month, where January = 1, February = 2, etc..
`Day`	Represents the day of the month (1 - 31).
`Hour`	Represents the hour of the day (0 - 23).
`Minute`	Represents the minute of the hour (0 - 59).
`Second`	Represents the second of the minute (0 - 59).

Table 4.65: FS_FILETIME - list of structure member variables

4.9.14 FS_SetBusyLEDCallback()

Description

Specifies callback function to control an LED which shows the state of a specific volume.

Prototype

```
void FS_SetBusyLEDCallback (const char *            sVolumeName,
                            FS_BUSY_LED_CALLBACK * pfBusyLEDCallback);
```

Parameter	Description
sVolumeName	Volume name as a string.
pfBusyLEDCallback	Pointer to a busy LED function.

Table 4.66: FS_SetBusyLEDCallback() parameter list

Additional Information

If you intend to show any volume read/write activity, use this function to set the busy indication for the desired volume.

Type `FS_BUSY_LED_CALLBACK` is defined as follows:

```
typedef void (FS_BUSY_LED_CALLBACK)(U8 OnOff);
```

The parameter `OnOff` indicates whether the LED should be switched on or off.

Example

```
#include "FS.h"

void SetBusyLED(U8 OnOff) {
  if (OnOff) {
    HW_SetLED();
  } else {
    HW_ClrLED();
  }
}

void MainTask(void) {
  FS_FILE * pFile;

  FS_Init();
  FS_SetBusyLEDCallback(“ram:”, &SetBusy);
  pFile = FS_FOpen(“ram:\\file.txt“, “w“);
  FS_FClose();
}
```

4.9.15 FS_SetVolumeLabel()

Description

Sets a label to a specific volume.

Prototype

```
int FS_SetVolumeLabel (const char * sVolume,
                       char  *      pVolumeLabel);
```

Parameter	Description
sVolume	Volume name as a string.
pVolumeLabel	Pointer to a buffer with the new volume label. NULL indicates, that the volume label should be deleted.

Table 4.67: FS_GetVolumeInfo() parameter list

Return Value

== 0: On Success.
==-1: In case of any error.

4.10 Storage layer functions

4.10.1 FS_STORAGE_GetDeviceInfo()

Description

Returns the device status.

Prototype

```
int FS_STORAGE_GetDeviceInfo (const char  * sVolume,
                              FS_DEV_INFO * pDevInfo);
```

Parameter	Description
sVolume	Name of the device to check.
pDeviceInfo	Pointer to a data structure of type `FS_DEV_INFO`.

Table 4.68: FS_STORAGE_GetDeviceInfo() parameter list

Return Value

==0: Ok
==-1: Device is not ready or a general error has occured.

4.10.2 FS_STORAGE_Init()

Description

This function only intializes the driver and OS if necessary.

Prototype

```
void FS_STORAGE_Init (void);
```

Return value

The return value is the number of drivers can be used at the same time. These number of drivers is relevant for the high-level initialzation function `FS_Init()`. `FS_Init()` uses these information to allocate the sector buffers which are neccessary for a file system operation.

Additional information

`FS_STORAGE_Init()` initializes the storage layer of a driver. If you use `FS_STORAGE_Init()` instead of FS_Init(), only the storage layer functions like `FS_STORAGE_ReadSector()` or `FS_STORAGE_WriteSector()` are available. This means that the file system can be used as a pure sector read/write software. This can be useful when using the file system as a USB mass storage client driver.

4.10.3 FS_STORAGE_ReadSector()

Description

Reads a sector from a device.

Prototype

```
int FS_STORAGE_ReadSector (const char * sVolume,
                           const void * pData,
                           U32          SectorIndex);
```

Parameter	Description
sVolume	Volume name. If not specified, the first device in the volume table will be used.
pData	Pointer to a buffer where the read data will be stored.
SectorIndex	Index of the sector from which data should be read.

Table 4.69: FS_STORAGE_ReadSector() parameter list

Return value

== 0: On success
!= 0: On error

4.10.4 FS_STORAGE_ReadSectors()

Description

Reads multiple sectors from a device.

Prototype

```
int FS_STORAGE_ReadSectors (const char * sVolume,
                            void *       pData,
                            U32          FirstSector,
                            U32          NumSectors);
```

Parameter	Description
sVolume	Volume name. If not specified, the first device in the volume table will be used.
pData	Pointer to a data buffer where the read data should be stored.
FirstSector	First sector to read.
NumSectors	Number of sectors which should be read.

Table 4.70: FS_STORAGE_ReadSectors() parameter list

Return value

== 0: On success
!= 0: On error

4.10.5 FS_STORAGE_Sync()

Description

Writes cached data to the storage medium and sends a command to the driver to finalize all pending tasks.

Prototype

```
void FS_STORAGE_Sync (const char * sVolume);
```

Parameter	Description
sVolume	Volume name. If not specified, the first device in the volume table will be used.

Table 4.71: FS_STORAGE_Sync() parameter list

4.10.6 FS_STORAGE_Unmount()

Description

Unmounts a given volume at the driver layer. The function also sends an unmount command to the driver, and marks the volume as unmounted and uninitialized.

Prototype

```
void FS_STORAGE_Unmount (const char * sVolume);
```

Parameter	Description
`sVolume`	`sVolume` is the name of a volume. If not specified, the first device in the volume table will be used.

Table 4.72: FS_STORAGE_Unmount() parameter list

4.10.7 FS_STORAGE_WriteSector()

Description

Writes one sector to a device.

Prototype

```
int FS_STORAGE_WriteSector (const char * sVolume,
                            const void * pData,
                            U32          SectorIndex);
```

Parameter	Description
sVolume	Volume name. If not specified, the first device in the volume table will be used.
pData	Pointer to the data which should be written to the device.
SectorIndex	Index of the sector to which data should be written.

Table 4.73: FS_STORAGE_WriteSector() parameter list

Return value

== 0: On success
!= 0: On error

4.10.8 FS_STORAGE_WriteSectors()

Description

Writes multiple sectors to a device.

Prototype

```
int FS_STORAGE_WriteSectors (const char * sVolume,
                             const void * pData,
                             U32          FirstSector,
                             U32          NumSectors)
```

Parameter	Description
sVolume	Volume name. If not specified, the first device in the volume table will be used.
pData	Pointer to the data which should be written to the device.
FirstSector	Start sector of the write operation.
NumSectors	Number of sectors that should be written.

Table 4.74: FS_STORAGE_WriteSectors() parameter list

Return value

== 0: On success
!= 0: On error

4.11 FAT related functions

4.11.1 FS_FAT_CheckDisk()

Description

Checks and repairs a FAT volume.

Prototype

```
int FS_FAT_CheckDisk (const char *       sVolumeName,
                      void *             pBuffer,
                      U32                BufferSize,
                      int                MaxRecursionLevel,
                      FS_QUERY_F_TYPE  * pfOnError);
```

Parameter	Description
`sVolumeName`	Volume name as a string.
`pBuffer`	Pointer to a buffer that will be used by `FS_FAT_CheckDisk()`.
`BufferSize`	Size of the specified buffer.
`MaxRecursionLevel`	The maximum directory recursion depth `FS_FAT_CheckDisk()` should check.
`pfOnError`	Pointer to a callback function for the error handling.

Table 4.75: FS_FAT_CheckDisk() parameter list

Return value

== 0: Ok
== 1: An error has be found and repaired, retry is required.
== 2: User specified an abort of checkdisk operation thru callback.

Additional Information

This function can be used to check if there are any errors on a specific volume and if necessary, repair the found error.

The buffer that `pBuffer` points to should be at least 4 Kbyte.
The minimum size of the buffer can be calculated as follows:

12 Bytes x (Bytes per sector x 8) / (Fat type),

where FAT type is either 12, 16, or 32.

The type `FS_QUERY_F_TYPE` is defined as follows:

```
typedef int (FS_QUERY_F_TYPE)(int ErrCode, ...);
```

The callback is used to notify the user about the error that occurred and to ask whether the error should be fixed. To get a detailed information string of the error that occurred, the parameter `ErrCode` can be passed to `FS_FAT_CheckDisk_ErrCode2Text()`.

Example

```
#include <stdarg.h>
#include "FS.h"

static U32 _aBuffer[5000];
/*********************************************************************
*
*       _OnError
*/
int _OnError(int ErrCode, ...) {
  va_list ParamList;
  const char * sFormat;
  char  c;
  char ac[1000];

  sFormat = FS_FAT_CheckDisk_ErrCode2Text(ErrCode);
  if (sFormat) {
    va_start(ParamList, ErrCode);
    vsprintf(ac, sFormat, ParamList);
    printf("%s\n", ac);
  }
  if (ErrCode != FS_ERRCODE_CLUSTER_UNUSED) {
    printf("  Do you want to repair this? (y/n/a) ");
  } else {
    printf("  * Convert lost cluster chain into file (y)\n"
           "  * Delete cluster chain                 (d)\n"
           "  * Do not repair                        (n)\n"
           "  * Abort                                (a) ");
    printf("\n");
  }
  c = getchar();
  printf("\n");
  if ((c == 'y') || (c == 'Y')) {
    return 1;
  } else if ((c == 'a') || (c == 'A')) {
    return 2;
  } else if ((c == 'd') || (c == 'D')) {
    return 3;
  }
  return 0;     // Do not fix.
}

/*********************************************************************
*
*       MainTask
*/
void MainTask(void) {
  FS_Init();
  while (FS_FAT_CheckDisk("", &_aBuffer[0], sizeof(_aBuffer), 5, _OnError) == 1) {
  }
}
```

4.11.2 FS_FAT_CheckDisk_ErrCode2Text()

Description

Returns an error string to a specific check-disk error code.

Prototype

```
const char * FS_FAT_CheckDisk_ErrCode2Text (int ErrCode);
```

Parameter	Description
ErrCode	Check-disk error code.

Table 4.76: FS_FAT_CheckDisk_ErrCode2Text() parameter list

Return value

The following error codes are defined as:

Code	Description
`FS_ERRCODE_0FILE`	A file of size zero has allocated cluster(s).
`FS_ERRCODE_SHORTEN_CLUSTER`	A cluster chain for a specific file is longer than its file size.
`FS_ERRCODE_CROSSLINKED_CLUSTER`	A cluster is cross-linked (used for multiple files / directories)
`FS_ERRCODE_FEW_CLUSTER`	Too few clusters allocated to file.
`FS_ERRCODE_CLUSTER_UNUSED`	A cluster is marked as used, but not assigned to a file or directory.
`FS_ERRCODE_CLUSTER_NOT_EOC`	A cluster is not marked as end-of-chain.
`FS_ERRCODE_INVALID_CLUSTER`	A cluster is not a valid cluster.

Table 4.77: FS_FAT_CheckDisk_ErrCode2Text() - list of error code values

Typically, this function is used in the callback function for the error handling that is used by `FS_FAT_CheckDisk()`. See “FS_FAT_CheckDisk()” on page 110 for an example.

4.11.3 FS_FormatSD()

Description

Performs a high-level format of a device according to the SD Specification - File system specification.

Prototype

```
int FS_FormatSD (const char * pVolumeName);
```

Parameter	Description
pVolumeName	Name of the device to format.

Table 4.78: FS_FormatSD() parameter list

Return value

== 0: File system has been started.
!= 0: An error has occurred.

Additional Information

For further information refer to SD Specification - Part 2 - File System Specification (May 9, 2006, *www.sdcard.org*).

4.11.4 FS_FAT_SupportLFN()

Note: The LFN package is needed to support long file names.

Description

Adds long file name support to the file system.

Prototype

```
void FS_SupportLFN(void);
```

Additional Information

The FAT file system was not designed for long file name (LFN) support, limiting names to twelve characters (8.3). LFN support may be added to any of the FAT file systems, but there are legal issues that must be settled with Microsoft before end applications make use of this feature. Long file names are inherent to this proprietary file system relieving it of any legal issues.

4.12 File system cache functions

4.12.1 FS_AssignCache()

Description

Adds a cache to a specific device.

Prototype

```
I32 FS_AssignCache (const char *         pDevName,
                    void *               pCacheData,
                    U32                  NumBytes
                    FS_INIT_CACHE *      pfInit);
```

Parameter	Description
`pDevName`	pDevName is the name of a device. If not specified, the first device in the volume table will be used.
`pCacheData`	Pointer to a buffer that should be used as cache.
`NumBytes`	Size of the specified buffer.

`pfInit`	Pointer to the initialization function of the desired cache module. The following values can be used:
	`FS_CACHE_ALL`
	`FS_CACHE_MAN`
	`FS_CACHE_RW`
	`FS_CACHE_RW_QUOTA`

Table 4.79: FS_AssignCache() parameter list

Return value

> 0: Buffer is used as cache for the specified device.
== 0: Buffer cannot be used as cache for this device.

Additional Information

To disable the cache for a specific device, call `FS_AssignCache()` with `NumBytes == 0`. In this case the return value will be 0.
There are four different available cache modules that can be assigned to a specific device. These modules are the following:

Cache module	Description
`FS_CACHE_ALL`	This module is a pure read cache. All sectors that are read from a volume are cached. This module does not need to be configured. Caching is enabled right after calling `FS_AssignCache()`.
`FS_CACHE_MAN`	This module is also a pure read cache. In contrast to the `FS_CACHE_ALL`, this module does only cache the management sector of a file system (for example FAT sectors). Caching is enabled right after calling `FS_AssignCache()`.
`FS_CACHE_RW`	`FS_CACHE_RW` is a configurable cache module. This module can be either used as read, write or as read/write cache. Additionally, the sectors that should be cached are also configurable. Refer to `FS_CACHE_SetMode()` to configure the `FS_CACHE_RW` module.
`FS_CACHE_RW_QUOTA`	`FS_CACHE_RW_QUOTA` is a configurable cache module. This module can be either used as read, write or as read/write cache. To configure the cache module properly, `FS_CACHE_SetMode()` and `FS_CACHE_SetQuota` need to be called. Otherwise the functionality inside the cache is disabled.

Example

```
#include "FS.h"

static char _acCache[100*1024]; /* Use a 100 Kbyte cache */

void Function(void) {
  /* Assign a cache to the first available device */
  FS_AssignCache("", _acCache, sizeof(_acCache), FS_CACHE_ALL);
  /* Do some work*/
  DoWork();
  /* Disable the read cache */
  FS_AssignCache("", NULL, 0);
}
```

4.12.2 FS_CACHE_Clean()

Description

Cleans a cache if sectors that are marked as dirty need to be written to the device.

Prototype

```
void FS_CACHE_Clean (const char * pDevName);
```

Parameter	Description
pDevName	pDevName is the name of a device. If not specified, the first device in the volume table will be used.

Table 4.80: FS_CACHE_Clean() parameter list

Additional Information

Because only write or read/write caches need to be cleaned, this function should be called for volumes where the `FS_CACHE_RW` module is assigned. The other cache modules ignore the cache clean operation.
Cleaning of the cache is also performed when the volume is unmounted through `FS_Unmount()` or disabling or reassigning the cache through `FS_AssignCache()`.

Example

```
#include "FS.h"

static char _acCache[100*1024]; /* Use a 100 Kbyte Cache */

void Function(void) {
  /* Assign a cache to the first available device */
  FS_AssignCache("", _acCache, sizeof(_acCache), FS_CACHE_ALL);
  /* Set the FS_CACHE_RW module to cache all sectors
   * Sectors are cached for read and write. Write back operation to volume
   * are delayed.
   */
  FS_CACHE_SetMode("", FS_SECTOR_TYPE_MASK_ALL, FS_CACHE_MODE_FULL);
  /* Do some work*/
  DoWork();
  FS_CACHE_Clean("");
  DoOtherWork();
  /* Disable cache */
  FS_AssignCache("", NULL, 0);
}
```

4.12.3 FS_CACHE_SetMode()

Description

Sets the mode for the cache.

Prototype

```
int FS_CACHE_SetMode (const char * pDevName,
                      int           TypeMask,
                      int           ModeMask);
```

Parameter	Description
pDevName	pDevName is the name of a device. If not specified, the first device in the volume table will be used.
TypeMask	Specifies the sector types that should be cached. This parameter can be an OR-combination of the following sector type mask.
ModeMask	Specifies the cache mode that should be used. Use one of the following parameters as cache mode mask.

Table 4.81: FS_CACHE_SetMode() parameter list

Permitted values for parameter TypeMask (OR-combinable)	
FS_SECTOR_TYPE_MASK_DATA	Caches all data sectors.
FS_SECTOR_TYPE_MASK_DIR	Caches all directory sectors.
FS_SECTOR_TYPE_MASK_MAN	Caches all management sectors.
FS_SECTOR_TYPE_MASK_ALL	Caches all sectors by an OR-combination of: FS_SECTOR_TYPE_MASK_DATA FS_SECTOR_TYPE_MASK_DIR FS_SECTOR_TYPE_MASK_MAN

Permitted values for parameter ModeMask (OR-combinable)	
FS_CACHE_MODE_R	Sectors of types defined in TypeMask are copied to cache when read from volume.
FS_CACHE_MODE_WT	Sectors of types defined in TypeMask are copied to cache and also written to the volume. (Writethrough cache)
FS_CACHE_MODE_WB	Sector types defined in TypeMask are lazily written back to the device. (Writeback cache)

Return value

== 0: Setting the mode of the cache module was successful.
== -1: Setting the mode of the cache module was not successful.

Additional Information

This function is only usable with the FS_CACHE_RW and FS_CACHE_RW_QUOTA module, after the FS_CACHE_RW cache has been assigned to a volume. The cache module needs to be configured with this function. Otherwise neither read nor write operations are cached.

Example

```
#include "FS.h"

static char _acCache[100*1024]; /* Use a 100 Kbyte cache */

void Function(void) {
  /* Assign a cache to the first available device */
  FS_AssignCache("", _acCache, sizeof(_acCache), FS_CACHE_RW);
  /* Set the FS_CACHE_RW module to cache all sectors
   * Sectors are cached for read and write. Write back operation to volume
   * are delayed.
   */
  FS_CACHE_SetMode("", FS_SECTOR_TYPE_MASK_ALL, FS_CACHE_MODE_WB);
  /* Do some work*/
  DoWork();
  FS_CACHE_Clean("");
  DoOtherWork();
  /* Disable cache */
  FS_AssignCache("", NULL, 0);
}
```

4.12.4 FS_CACHE_SetQuota()

Description

Sets the quotas for the different sector types in the `CacheRW_Quota` cache module.

Prototype

```
int FS_CACHE_SetMode (const char * pDevName,
                      int          TypeMask,
                      U32          NumSectors);
```

Parameter	Description
`pDevName`	`pDevName` is the name of a device. If not specified, the first device in the volume table will be used.
`TypeMask`	Specifies the sector types that should be cached. This parameter can be an OR-combination of the following sector type mask.
`NumSectors`	Specifies the number of sectors each sector type that is defined by `TypeMask` should reserve.

Table 4.82: FS_CACHE_SetQuota() parameter list

Permitted values for parameter `TypeMask` (`OR`-combinable)	
`FS_SECTOR_TYPE_MASK_DATA`	Caches all data sectors.
`FS_SECTOR_TYPE_MASK_DIR`	Caches all directory sectors.
`FS_SECTOR_TYPE_MASK_MAN`	Caches all management sectors.

`FS_SECTOR_TYPE_MASK_ALL`	All sectors are cached.
	This is an OR-combination of
	`FS_SECTOR_TYPE_MASK_DATA`
	`FS_SECTOR_TYPE_MASK_DIR`
	`FS_SECTOR_TYPE_MASK_MAN`

Return value

== -1: Setting the quota of the cache module was not successful.
== 0: Setting the quota of the cache module was successful.

Additional Information

This function is currently only usable with the `FS_CACHE_RW_QUOTA` module. After the `FS_CACHE_RW_QUOTA` cache has been assigned to a volume and the cache mode has been set, the quotas for the different sector types need to be configured with this function. Otherwise neither read nor write operations are cached.

Example

```
#include "FS.h"

static char _acCache[100*1024]; /* Use a 100 Kbyte cache */

void Function(void) {
  /* Assign a cache to the first available device */
  FS_AssignCache("", _acCache, sizeof(_acCache), FS_CACHE_RW_QUOTA);
  /* Set the FS_CACHE_RW module to cache all sectors
   * Sectors are cached for read and write. Write back operation to volume
   * are delayed.
   */
  FS_CACHE_SetMode("", FS_SECTOR_TYPE_MASK_ALL, FS_CACHE_MODE_FULL);
  /* Set the quotas for directory and data sector types
   * in the CACHE_RW_QUOTA module to 10 sectors each
   */
  FS_CACHE_SetQuota("", FS_SECTOR_TYPE_MASK_DATA | FS_SECTOR_TYPE_MASK_DIR, 10);
  /* Do some work*/
  DoWork();
  FS_CACHE_Clean("");
  DoOtherWork();
  /* Disable cache */
  FS_AssignCache("", NULL, 0);
}
```

4.13 Error handling functions

4.13.1 FS_ClearErr()

Description

Clears the error status of a file.

Prototype

```
void FS_ClearErr (FS_FILE * pFile);
```

Parameter	Description
pFile	Pointer to a data structure of type `FS_FILE`.

Table 4.83: FS_ClearErr() parameter list

Additional Information

This routine should be called after you have detected an error so that you can check for success of the next file operations.

Example

```
void MainTask(void) {
  FS_FILE *pFile;
  int Err;

  pFile = FS_FOpen("test.txt", "r");
  if (pFile != 0) {
    Err = FS_FError(pFile);
    if (Err != FS_ERR_OK) {
      FS_ClearErr(pFile);
    }
    FS_FClose(pFile);
  }
}
```

4.13.2 FS_FEof()

Description

Tests for end-of-file on a given file pointer.

Prototype

```
int FS_FEof (FS_FILE * pFile);
```

Parameter	Description
pFile	Pointer to a data structure of type `FS_FILE`.

Table 4.84: FS_FEof() parameter list

Return value

== 0: If the end of file has not been reached.
== 1: If the end of file has been reached.

Additional Information

The `FS_FEof` function determines whether the end of a given file pointer has been reached. When end of file is reached, read operations return an end-of-file indicator until the file pointer is closed or until `FS_FSeek`, or `FS_ClearErr` is called against it.

Example

```
int ReadFile(FS_File * pFile, char * pBuffer, int NumBytes) {
  FS_FILE * pFile;
  char      acBuffer[100];
  char      acLog[100];
  int       Count;
  int       Total;
  I16       Error;

  Total = 0;
  pFile = FS_FOpen("default.txt", "r");
  if (pFile == NULL) {
    FS_X_ErrorOut("Could not open file.");'
  }
  /* Cycle until end of file reached: */
  while (!FS_FEof(pFile)) {
    Count = FS_Read(pFile, &acBuffer[0], sizeof(acBuffer));
    Error = FS_FError(pFile);
    if (Error) {
      sprintf(acLog, "Could not read from file:\nReason = %s",
              FS_ErrorNo2Text(Error));
      FS_X_ErrorOut(acLog);
      break;
    }
    /* Total up actual bytes read */
    Total += Count;
  }
  sprintf(acLog, "Number of read bytes = %d\n", Total);
  FS_X_Log(acLog);
  FS_FClose(pFile);
}
```

4.13.3 FS_FError()

Description

Returns the current error status of a file.

Prototype

```
I16 FS_FError (FS_FILE * pFile);
```

Parameter	Description
pFile	Pointer to a data structure of type `FS_FILE`.

Table 4.85: FS_FError() parameter list

Return value

`FS_ERR_OK` if no errors.
A value not equal to `FS_ERR_OK` if a file operation caused an error.

Additional Information

The return value is not `FS_ERR_OK` only when a file operation caused an error and the error was not cleared by calling `FS_ClearErr()` or any other operation that clears the previous error status.
The following error codes are available:

Code	Description
FS_ERR_OK	No error.
FS_ERR_EOF	End-of-file has been reached.
FS_ERR_DISKFULL	Unable to write data because there is no more space on the media.
FS_ERR_INVALIDPAR	An µC/FS function has been called with an illegal parameter.
FS_ERR_WRITEONLY	A read operation has been made on a file open for writing only.
FS_ERR_READONLY	A write operation has been made on a file open for reading only.
FS_ERR_READERROR	An error occurred during a read operation.
FS_ERR_WRITEERROR	An error occurred during a write operation.
FS_ERR_DISKCHANGED	Media has been changed, although the file was still open.
FS_ERR_CLOSE	An error occurred during the close operation.

Table 4.86: FS_FError() - list of error code values

Example

```
void MainTask(void) {
  FS_FILE *pFile;

  pFile = FS_FOpen("test.txt", "r");
  if (pFile != 0) {
    I16 Err;
    Err = FS_FError(pFile);
    FS_FClose(pFile);
  }
}
```

4.13.4 FS_ErrorNo2Text()

Description

Retrieves text for a given error code.

Prototype

```
const char * FS_ErrorNo2Text (int ErrCode);
```

Parameter	Description
ErrCode	The returned error code.

Table 4.87: FS_ErrorNo2Text() parameter list

Return value

Returns the string according to the ErrCode.

Example

```
void MainTask(void) {
  FS_FILE *pFile;

  pFile = FS_FOpen("test.txt", "r");
  if (pFile != 0) {
    int Err;
    Err = FS_FError(pFile);
    FS_X_Log("Open file error: ");
    FS_X_Log(FS_ErrorNo2Text(Err));
    FS_FClose(pFile);
  }
}
```

4.14 Obsolete functions

This section contains reference information for obsolete functions.

4.14.1 FS_CloseDir()

Description

Closes a directory referred to by the parameter `pDir`.

Prototype

```
int FS_CloseDir (FS_DIR * pDir);
```

Parameter	Description
`pDir`	Pointer to a data structure of type `FS_DIR`.

Table 4.88: FS_CloseDir() parameter list

Return Value

== 0: If the directory was successfully closed.
== -1: In case of any error.

Example

```
void MainTask(void) {
  FS_DIR    *pDir;
  FS_DIRENT *pDirEnt;

  pDir = FS_OpenDir("");          /* Open the root directory of default device */
  if (pDir) {
    do {
      char acDirName[20];
      pDirEnt = FS_ReadDir(pDir);
      FS_DirEnt2Name(pDirEnt, acDirName); /* Get name of the current DirEntry */
      if ((void*)pDirEnt == NULL) {
        break;                            /* No more files or directories */
      }
      sprintf(_acBuffer,"  %s\n", acName);
      FS_X_Log(_acBuffer);
    } while (1);
    FS_CloseDir(pDir);
  } else {
    FS_X_ErrorOut("Unable to open directory\n");
}
```

4.14.2 FS_DirEnt2Attr()

Description

Retrieves the attributes of the directory entry referred to by pDirEnt.

Prototype

```
void FS_DirEnt2Attr (FS_DIRENT * pDirEnt,
                     U8 *        pAttr);
```

Parameter	Description
pDirEnt	Pointer to a directory entry, read by FS_ReadDir().
pAttr	Pointer to U8 variable in which the attributes should be stored.

Table 4.89: FS_DirEnt2Attr() parameter list

Additional Information

These attributes are available:

Parameter	Description
FS_ATTR_DIRECTORY	pDirEnt is a directory.
FS_ATTR_ARCHIVE	pDirEnt has the ARCHIVE attribute set.
FS_ATTR_READ_ONLY	pDirEnt has the READ ONLY attribute set.
FS_ATTR_HIDDEN	pDirEnt has the HIDDEN attribute set.
SYSTEM	pDirEnt has the SYSTEM attribute set.

Table 4.90: FS_DirEnt2Attr() - list of possible attributes

pDirEnt should point to a valid FS_DIRENT structure. FS_DirEnt2Attr() checks if the pointer is valid. To get a valid pointer, FS_ReadDir() should be called before using FS_DirEnt2Attr(). Refer to “FS_ReadDir()” on page 135 for more information.

Example

```
void MainTask(void) {
  FS_DIR    *pDir;
  FS_DIRENT *pDirEnt;
  char acBuffer[200];
  pDir = FS_OpenDir("");       /* Open root directory of default device */
  if (pDir) {
    do {
      char acName[20];
      U8 Attr;
      pDirEnt = FS_ReadDir(pDir);
      FS_DirEnt2Name(pDirEnt, acName);
      FS_DirEnt2Attr(pDirEnt, &Attr);
      if ((void*)pDirEnt == NULL) {
        break;               /* No more files */
      }
      sprintf(_acBuffer,"  %s %s Attributes: %s%s%s%s\n", acName,
                        (Attr & FS_ATTR_DIRECTORY) ? "(Dir)" : "      ",
                        (Attr & FS_ATTR_ARCHIVE)   ? "A"     : "-",
                        (Attr & FS_ATTR_READ_ONLY) ? "R"     : "-",
                        (Attr & FS_ATTR_HIDDEN)    ? "H"     : "-",
                        (Attr & FS_ATTR_SYSTEM)    ? "S"     : "-");
      FS_X_Log(acBuffer);
    } while (1);
    FS_CloseDir(pDir);
  } else {
    FS_X_ErrorOut("Unable to open directory\n");
  }
}
```

4.14.3 FS_DirEnt2Name()

Description

Retrieves the name of the directory entry referred to by pDirEnt.

Prototype

```
void FS_DirEnt2Name (FS_DIRENT * pDirEnt,
                     char *      pBuffer);
```

Parameter	Description
pDirEnt	Pointer to a directory entry, read by FS_ReadDir().
pBuffer	Pointer to the buffer that will receive the text.

Table 4.91: FS_DirEnt2Name() parameter list

Additional Information

If pDirEnt and pBuffer are valid, the name of the directory is copied to the buffer that pBuffer points to. Otherwise pBuffer is NULL.
pDirEnt should point to a valid FS_DIRENT structure. FS_DirEnt2Name() checks if the pointers are valid. To get a valid pointer, FS_ReadDir() should be called before using FS_DirEnt2Name(), otherwise pBuffer is NULL. Refer to "FS_ReadDir()" on page 135 for more information.

Example

```
void MainTask(void) {
  char acDirName[20];
  FS_DIR    *pDir ;
  FS_DIRENT *pDirEnt ;

  pDir = FS_OpenDir("");      /* Open root directory of default device */
  pDirEnt = FS_ReadDir(pDir); /* Read the first directory entry */
  FS_DirEnt2Name(pDirEnt, acDirName);
  FS_X_Log(acDirName);
}
```

4.14.4 FS_DirEnt2Size()

Description

Returns the size in bytes of the directory entry referred to pDirEnt.

Prototype

```
U32 FS_DirEnt2Size (FS_DIRENT * pDirEnt);
```

Parameter	Description
pDirEnt	Pointer to a directory entry, read by FS_ReadDir().

Table 4.92: FS_DirEnt2Size() parameter list

Return value

File size in bytes.
0 in case of any error.

Additional Information

If pDirEnt is valid, the size of the directory entry will be returned. Otherwise the return value is 0.
pDirEnt should point to a valid FS_DIRENT structure. FS_DirEnt2Name() checks if the pointers are valid. To get a valid pointer, FS_ReadDir() should be called before using FS_DirEnt2Size(). Refer to "FS_ReadDir()" on page 135 for more information.

Example

```
void MainTask(void) {
  U32       FileSize;
  FS_DIR    *pDir ;
  FS_DIRENT *pDirEnt ;

  pDir = FS_OpenDir("");       /* Open root directory of default device */
  pDirEnt = FS_ReadDir(pDir); /* Read the first directory entry */
  FileSize = FS_DirEnt2Size(pDirEnt);
  if (FileSize) {
    char ac[50] ;
    sprintf(ac, "File size = %lu\n", FileSize);
    FS_X_Log(ac) ;
}
```

4.14.5 FS_DirEnt2Time()

Description

Returns the timestamp of the directory entry referred to by pDirEnt.

Prototype

```
U32 FS_DirEnt2Size (FS_DIRENT * pDirEnt);
```

Parameter	Description
pDirEnt	Pointer to a directory entry, read by FS_ReadDir().

Table 4.93: FS_DirEnt2Time() parameter list

Return value

The timestamp of the current directory entry.

Additional Information

If pDirEnt is valid, the timestamp of the directory entry will be returned. Otherwise, the return value is 0.
pDirEnt should point to a valid FS_DIRENT structure. FS_DirEnt2Name() checks if the pointer is valid. To get a valid pointer, FS_ReadDir() should be called before using FS_DirEnt2Size(). Refer to "FS_ReadDir()" on page 135 for more information.
A timestamp is a packed value with the following format.

Bits	Description
0-4	Second divided by 2
5-10	Minute (0 - 59)
11-15	Hour (0-23)
16-20	Day of month (1-31)
21-24	Month (January -> 1, February -> 2, etc.)
25-31	Year offset from 1980. Add 1980 to get current year.

Table 4.94: FS_DirEnt2Time() - timestamp format description

To convert a timestamp to a FS_FILETIME structure, use the function FS_TimeStampToFileTime().

Example

```
void MainTask(void) {
  U32         TimeStamp;
  FS_DIR     * pDir ;
  FS_DIRENT * pDirEnt ;
  char         acLog[100] ;
  char         acFileName[40];
  FS_FILETIME FileTime;

  pDir       = FS_OpenDir("");       /* Open root directory of default device */
  pDirEnt    = FS_ReadDir(pDir);     /* Read the first directory entry */
  FS_DirEnt2Name(pDirEnt, &acFileName[0]);
  TimeStamp = FS_DirEnt2Time(pDirEnt);
  FS_TimeStampToFileTime(&TimeStamp, &FileTime);
  sprintf(ac, "File time of %s: %d-.2d-%.2d %.2d:%.2d:%.2d",
          acFileName,
          FileTime.Year, FileTime.Month,  FileTime.Day,
          FileTime.Hour, FileTime.Minute, FileTime.Second);
  FS_X_Log(ac);
}
```

4.14.6 FS_GetDeviceInfo()

Description

Returns the device status.

Prototype

```
int FS_GetDeviceInfo(const char  * sVolume,
                     FS_DEV_INFO * pDevInfo);
```

Parameter	Description
sVolume	Name of the device to check.
pDeviceInfo	Pointer to a data structure of type FS_DEV_INFO.

Table 4.95: FS_GetDeviceInfo() parameter list

Additional information

This function is obsolete. Use instead “FS_STORAGE_GetDeviceInfo()” on page 102.

Return Value

==0: Ok
==-1: Device is not ready or a general error has occured.

4.14.7 FS_GetNumFiles()

Description

Returns the number of files in a directory opened by `FS_OpenDir()`.

Prototype

```
U32 FS_GetNumFiles (FS_DIR * pDir);
```

Parameter	Description
`pDir`	Pointer to a `FS_FILE` data structure.

Table 4.96: FS_GetNumFiles() parameter list

Return value

Number of files in a directory.
0xFFFFFFFF as return value indicates an error.

Additional Information

If `pDir` is valid, the number of files in the directory will be returned. To get a valid pointer, `FS_OpenDir()` should be called before using `FS_GetNumFiles()`. Refer to "FS_OpenDir()" on page 134 for more information.

Example

```
void NumFilesInDirectory(void) {
  U32    NumFilesInDir;
  FS_DIR    *pDir ;

  pDir = FS_OpenDir("");      /* Open root directory of default device */
  NumFilesInDir = FS_GetNumFiles(pDir);
  if (NumFilesInDir) {
    char ac[50] ;
    sprintf(ac, "NumFilesInDir = %lu\n", NumFilesInDir);
    FS_X_Log(ac) ;
  }
}
```

4.14.8 FS_InitStorage()

Description

This function only intializes the driver and OS if necessary.

Prototype

```
void FS_InitStorage (void);
```

Return value

The return value indicates the high level init how many drivers can be used at the same time. The function will accordingly allocate the sector buffers that are neccessary for a file system operation.

Additional information

If `FS_InitStorage()` is used to initialize a driver only the hardware layer functions `FS_ReadSector()`, `FS_WriteSector()`, and `FS_GetDeviceInfo()` are available.

This function is obsolete. Use instead “FS_STORAGE_Init()” on page 103.

4.14.8.1 FS_OpenDir()

Description

Opens an existing directory for reading.

Prototype

```
FS_DIR *FS_OpenDir (const char * pDirname);
```

Parameter	Description
pDirName	Fully qualified directory name.

Table 4.97: FS_OpenDir() parameter list

Return value

Returns the address of an `FS_DIR` data structure if the directory was opened.
In case of any error the return value is 0.

Additional Information

A fully qualified directory name looks like:

`[DevName:[UnitNum:]][DirPathList]DirectoryName`

where:

- `DevName` is the name of a device, for example `"ram"` or `"mmc"`. If not specified, the first device in the device table will be used.
  `UnitNum` is the number for the unit of the device. If not specified, unit 0 will be used. Note that it is not allowed to specify `UnitNum` if `DevName` has not been specified.
- `DirPathList` is a complete path to an existing subdirectory. The path must start and end with a '\' character. Directory names in the path are separated by '\'. If `DirPathList` is not specified, the root directory on the device will be used.
- `DirectoryName` and all other directory names have to follow the standard FAT naming conventions (for example 8.3 notation), if support for long file names is not enabled.

To open the root directory, simply use an empty string for `pDirName`.

Example

```
FS_DIR *pDir;

void FSTask1(void) {
  /* Open directory test - default driver on default device */
  pDir = FS_OpenDir("test");
}

void FSTask2(void) {
  /* Open root directory - RAM device driver on default device */
  pDir = FS_OpenDir("ram:");
}
```

4.14.8.2 FS_ReadDir()

Description

Reads next directory entry in directory specified by `pDir`.

Prototype

```
FS_DIRENT *FS_ReadDir (FS_DIR * pDir);
```

Parameter	Description
`pDir`	Pointer to an opened directory.

Table 4.98: FS_ReadDir() parameter list

Return value

Returns a pointer to a directory entry.
If there are no more entries in the directory or in case of any error, 0 is returned.

Example

Refer to “FS_CloseDir()” on page 126.

4.14.9 FS_ReadSector()

Description

Reads a sector from a device.

Prototype

```
int FS_ReadSector(const char *sVolume, const void *pData, U32 SectorIndex);
```

Parameter	Description
`sVolume`	Volume name.
`pData`	Pointer to a data buffer where the read data should be stored.
`SectorIndex`	Index of the sector from which data should be read.

Table 4.99: FS_ReadSector() parameter list

Return value

== 0: On success
!= 0: On error

Additional information

This function is obsolete. Use instead “FS_STORAGE_ReadSector()” on page 104.

4.14.9.1 FS_RewindDir()

Description

Sets the current pointer for reading a directory entry to the first entry in the directory.

Prototype

```
void FS_RewindDir (FS_DIR * pDir);
```

Parameter	Description
pDir	Pointer to directory structure.

Table 4.100: FS_RewindDir() parameter list

Example

```
void MainTask(void) {
  FS_DIR    *pDir;
  FS_DIRENT *pDirEnt;
  char acDirName[20];

  pDir = FS_OpenDir("");          /* Open the root directory of default device */
  if (pDir) {
    do {
      char acDirName[20];
      pDirEnt = FS_ReadDir(pDir);
      FS_DirEnt2Name(pDirEnt, acDirName);  /* Get name of the current DirEntry */
      if ((void*)pDirEnt == NULL) {
        break;                                 /* No more files or directories */
      }
      sprintf(_acBuffer,"  %s\n", acName);
      FS_X_Log(_acBuffer);
    } while (1);
    /* rewind to 1st entry */
    FS_RewindDir(dirp);
    /* display directory  again */
    do {
      pDirEnt = FS_ReadDir(pDir);
      FS_DirEnt2Name(pDirEnt, acDirName); /* Get name of the current DirEntry */
      if ((void*)pDirEnt == NULL) {
        break;                                /* No more files or directories */
      }
      sprintf(_acBuffer,"  %s\n", acName);
      FS_X_Log(_acBuffer);
    } while (1);
    FS_CloseDir(pDir);
  }
  else {
    FS_X_ErrorOut("Unable to open directory\n");
  }
}
```

4.14.10 FS_UnmountLL()

Description

Unmounts a given volume at driver layer. Sends an unmount command to the driver, marks the volume as unmounted and uninitialized.

Prototype

```
void FS_Unmount (const char * sVolume);
```

Parameter	Description
sVolume	sVolume is the name of a volume. If not specified, the first device in the volume table will be used.

Table 4.101: FS_UnmountLL() parameter list

Additional information

This function is obsolete. Use instead “FS_STORAGE_Init()” on page 103.

4.14.11 FS_WriteSector()

Description

Writes a sector to a device.

Prototype

```
int FS_WriteSector(const char *sVolume, const void *pData, U32 SectorIndex);
```

Parameter	Description
`sVolume`	Volume name.
`pData`	Pointer to the data which should be written to the device.
`SectorIndex`	Index of the sector to which data should be written.

Table 4.102: FS_WriteSector() parameter list

Return value

== 0: On success
!= 0: On error

Additional information

This function is obsolete. Use instead FS_STORAGE_Init() on page 104.

Chapter 5

Device drivers

µC/FS has been designed to cooperate with any kind of hardware. To use specific hardware with µC/FS, a so-called device driver for that hardware is required. The device driver consists of basic I/O functions for accessing the hardware and a global table that holds pointers to these functions.

5.1 General information

5.1.1 Default device driver names

By default the following identifiers are used for each driver.

Driver (Device)	Identifier	Name
Harddisk/CompactFlash	`FS_IDE_Driver`	"ide:"
MMC/SD SPI mode	`FS_MMC_SPI_Driver`	"mmc:"
MMC/SD Card mode	`FS_MMC_CardMode_Driver`	"mmc:"
NAND flash	`FS_NAND_Driver`	"nand:"
NOR flash	`FS_NOR_Driver`	"nor:"
RAM disk	`FS_RAMDISK_Driver`	"ram:"
WINDrive	`FS_WINDRIVE_Driver`	"win:"

Table 5.1: List of default device driver labels

To add a driver to µC/FS, `FS_AddDevice()` should be called with the proper identifier to mount the device driver to µC/FS before accessing the device or its units. Refer to "FS_AddDevice()" on page 44 for detailed information.

5.1.2 Unit number

Most driver functions as well as most of the underlying hardware functions receive the unit number as the first parameter. The unit number allows differentiation between the different instances of the same device types. If there are for example 2 NAND flashes which operate as 2 different devices, the first one is identified as Unit = 0, the second one as Unit = 1. If there is just a single instance (as in most systems), the unit parameter can be ignored by the underlying hardware functions.

5.2 RAM disk driver

µC/FS comes with a simple RAM disk driver that makes it possible to use a portion of your system RAM as drive for data storage. This can be very helpful to examine your system performance and may also be used as a in-system test procedure.

5.2.1 Supported hardware

The RAM driver can be used with every target with enough RAM. The size of the disk is defined as the number of sectors reserved for the drive.

5.2.2 Theory of operation

A RAM disk is a portion of memory that you allocate to use as a partition. The RAM disk driver takes some of your memory and pretends that it is a hard drive that you can format, mount, save files to, etc.

Remember that every bit of RAM is important for the well being of your system and the bigger your RAM disk is, the less memory there is available for your system.

5.2.3 Fail-safe operation

When power is lost, the data of the RAM drive is typically lost as well except for systems with Battery backup for the RAM used as storage device.

For this reason, fail safety is relevant only for systems which provide such battery backup.

Unexpected Reset

In case of an unexpected reset the data will be preserved. However, if the Power failure / unexpected Reset interrupts a write operation, the data of the sector may contain partially invalid data.

Power failure

Power failure causes an unexpected reset and has the same effects.

5.2.4 Wear leveling

The RAM disk driver does not require wear leveling.

5.2.5 Configuring the driver

5.2.5.1 Adding the driver to µC/FS

To add the driver, use `FS_AddDevice()` with the driver label `FS_RAMDISK_Driver`. This function has to be called from within `FS_X_AddDevices()`. Refer to "FS_X_AddDevices()" on page 262 for more information. Refer to "FS_X_AddDevices()" on page 262 for more information.

Example

```
FS_AddDevice(&FS_RAMDISK_Driver);
```

5.2.5.2 FS_RAMDISK_Configure()

Description

Configures a single RAM disk instance. This function has to be called from within `FS_X_AddDevices()` after adding an instance of the `RAMDisk` driver. Refer to "FS_X_AddDevices()" on page 262 for more information.

Prototype

```
void FS_RAMDISK_Configure(U8     Unit,
                          void * pData,
                          U16    BytesPerSector,
                          U32    NumSectors);
```

Parameter	Description
`Unit`	Unit number (0...N).
`pData`	Pointer to a data buffer.
`BytesPerSector`	Number of bytes per sector.
`NumSectors`	Number of sectors.

Table 5.2: FS_RAMDISK_Configure() parameter list

Additional information

The size of the disk is defined as the number of sectors reserved for the drive. Each sector consists of 512 bytes. The minimum value for `NumSectors` is 7. `BytesPerSector` defines the size of each sector on the RAM disk. A FAT file system needs a minimum sector size of 512 bytes.

Example

```
/*********************************************************************
*
*       FS_X_AddDevices
*
*  Function description
*    This function is called by the FS during FS_Init().
*    It is supposed to add all devices, using primarily FS_AddDevice().
*
*  Note
*    (1) Other API functions
*        Other API functions may NOT be called, since this function is called
*        during initialisation. The devices are not yet ready at this point.
*/
void FS_X_AddDevices(void) {
  void  * pRamDisk;

  //
  // Allocate memory for the RAM disk
  //
  pRamDisk = FS_Alloc(RAMDISK_NUM_SECTORS * RAMDISK_BYTES_PER_SECTOR);
  //
  // Add driver
  //
  FS_AddDevice(&FS_RAMDISK_Driver);
  //
  // Configure driver
  //
  FS_RAMDISK_Configure(0, pRamDisk, RAMDISK_BYTES_PER_SECTOR, RAMDISK_NUM_SECTORS);
}
```

5.2.6 Hardware functions

The RAM disk driver does not need any hardware function.

5.2.7 Addition information

5.2.7.1 Formatting

A RAM disk is unformatted after each startup. Exceptions from this rule are RAM disks, which are memory backed up with a battery.

You have to format every unformatted RAM disk with the `FS_Format()` function, before you can store data on it. If you use only one RAM disk in your application `FS_FORMAT()` can be called with an empty string as device name. For example, `FS_Format("", NULL);`

If you use more then one RAM disk, you have to specify the device name. For example, `FS_FORMAT("ram:0:", NULL);` for the first device and `FS_FORMAT("ram:1:", NULL);` for the second. Refer to "FS_Format()" on page 84 for more detailed information about the high-level format function of µC/FS.

5.3 NAND flash driver

µC/FS supports the use of NAND flashes. An optional driver for NAND flashes is available. The NAND driver requires very little RAM, it can work with sector sizes of 512 bytes or 2 Kbytes (small sectors even on large page NAND flashes) and is extremly efficient. The driver is designed to support one or multiple SLC (Single Level Cell) NAND flashes. The NAND flash driver can also be used to access ATMEL's DataFlash chips. To use it in your system, you will have to provide basic I/O functions for accessing your flash device.

This section first describes which devices are supported and describes all hardware access functions required by the NAND flash driver.

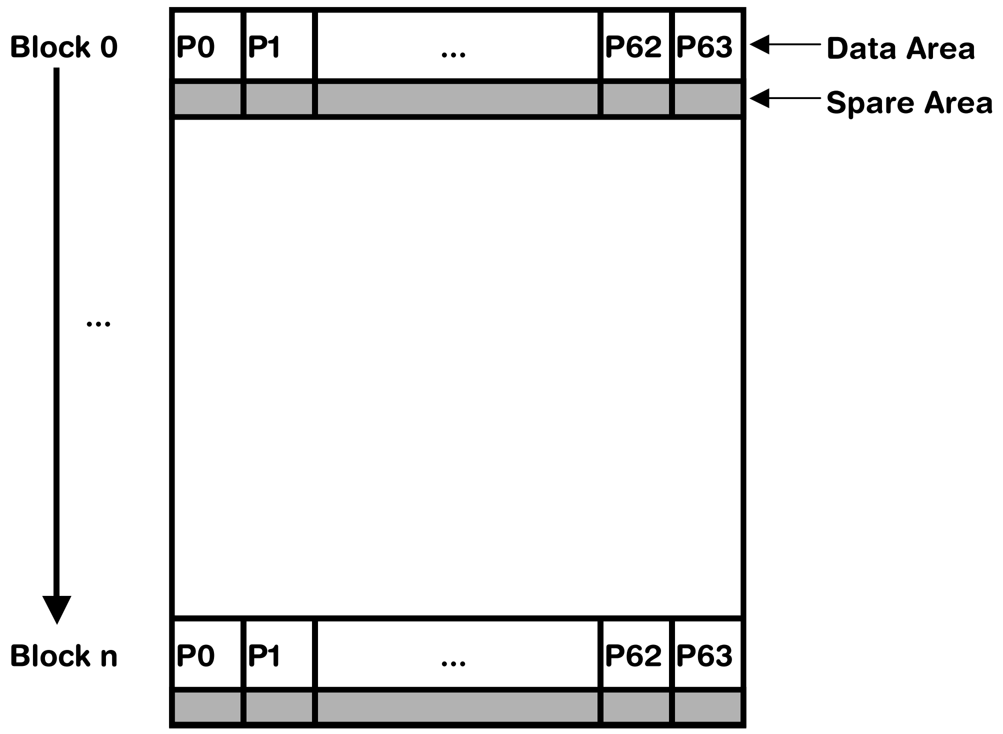
NAND flash organization

A NAND flash is a serial-type memory device which utilizes the I/O pins for both address and data input/output as well as for command inputs. The erase and program operations are automatically executed. To store data on the NAND flash device, it has to be low-level formatted.

NAND flashes consist of a number of blocks. Every block contains a number of sectors, typically 64. The sectors can be written to individually, one at a time. When writing to a sector, bits can only be written from 1 to 0. Only whole blocks (all sectors in the block) can be erased. Erasing means bringing all memory bits in all sectors of the block to logical 1.

Small NAND flashes (up to 256 Mbytes) have a page size of 528 bytes, 512 for data + 16 spare bytes for storing relevant information (ECC, etc.) to the page. Large NAND devices (256 Mbytes or more) have a page size of 2112 bytes, 2048 bytes for data + 64 bytes for storing relevant information to the page.

NAND flash organization



For example, a typical NAND flash with a size of 256 Mbytes has 2048 blocks of 64 pages of 2112 bytes (2048 bytes for data + 64 bytes).

5.3.1 Supported hardware

5.3.1.1 Tested and compatible NAND flashes

In general, the driver supports almost all Single-Level Cell NAND flashes (SLC). This includes NAND flashes with page sizes of 512+16 and 2048+64 bytes.

The table below shows the NAND flashes that have been tested or are compatible with a tested device:

Manufacturer	Device	Page size [Bytes]	Size [Bits]
Hynix	HY27xS08281A	512+16	16Mx8
	HY27xS08561M	512+16	32Mx8
	HY27xS08121M	512+16	64Mx8
	HY27xA081G1M	512+16	128Mx8
Samsung	K9F6408Q0xx	512+16	8Mx8
	K9F6408U0xx	512+16	8Mx8
	K9F2808Q0xx	512+16	16Mx8
	K9F2808U0xx	512+16	16Mx8
	K9F5608Q0xx	512+16	32Mx8
	K9F5608D0xx	512+16	32Mx8
	K9F5608U0xx	512+16	32Mx8
	K9F1208Q0xx	512+16	64Mx8
	K9F1208D0xx	512+16	64Mx8
	K9F1208U0xx	512+16	64Mx8
	K9F1208R0xx	512+16	64Mx8
	K9K1G08R0B	512+16	128Mx8
	K9K1G08B0B	512+16	128Mx8
	K9K1G08U0B	512+16	128Mx8
	K9K1G08U0M	512+16	128Mx8
	K9T1GJ8U0M	512+16	128Mx8
ST-Microelectronics	NAND128R3A	512+16	16Mx8
	NAND128W3A	512+16	16Mx8
	NAND256R3A	512+16	32Mx8
	NAND256W3A	512+16	32Mx8
	NAND512R3A	512+16	64Mx8
	NAND512W3A	512+16	64Mx8
	NAND01GR3A	512+16	128Mx8
	NAND01GW3A	512+16	128Mx8
Toshiba	TC5816BFT	512+16	2Mx8
	TC58V32AFT	512+16	4Mx8
	TC58V64BFTx	512+16	8Mx8
	TC58256AFT	512+16	32Mx8
	TC582562AXB	512+16	32Mx8
	TC58512FTx	512+16	64Mx8
	TH58100FT	512+16	256Mx8

Table 5.3: List of supported NAND flashes

Manufacturer	Device	Page size [Bytes]	Size [Bits]
Hynix	HY27UF082G2M	2048+64	256Mx8
	HY27UF084G2M	2048+64	512Mx8
	HY27UG084G2M	2048+64	512Mx8
	HY27UG084GDM	2048+64	512Mx8
Micron	MT29F2G08A	2048+64	256Mx8
	MT29F2G08B	2048+64	256Mx8
	MT29F4G08A	2048+64	512Mx8
	MT29F4G08B	2048+64	512Mx8
	MT29F2G16A	2048+64	128Mx16
Samsung	K9F1G08x0A	2048+64	256Mx8
	K9F2G08U0M	2048+64	256Mx8
	K9K2G08R0A	2048+64	256Mx8
	K9K2G08U0M	2048+64	256Mx8
	K9F4G08U0M	2048+64	512Mx8
	K9F8G08U0M	2048+64	1024Mx8
ST-Microelectronics	NAND01GR3B	2048+64	128Mx8
	NAND01GW3B	2048+64	128Mx8
	NAND02GR3B	2048+64	256Mx8
	NAND02GW3B	2048+64	256Mx8
	NAND04GW3	2048+64	512Mx8

Table 5.3: List of supported NAND flashes

Support for devices not in this list

Most other NAND flash devices are compatible with one of the supported devices. Thus, the driver can be used with these devices or may only need a little modification, which can be easily done. Get in touch with us, if you have questions about support for devices not in this list.

5.3.1.2 Tested and compatible DataFlash chips

The NAND flash driver fully supports the ATMEL DataFlash®/DataFlash Cards series up to 32 MBit. Currently the following devices are supported:

Manufacturer	Device
ATMEL	AT45DB011B
	AT45DB021B
	AT45DB041B
	AT45DB081B
	AT45DB161B
	AT45DB321C
	AT45BR3214B
	AT45DCB002
	AT45DCB004

Table 5.4: List of supported serial flash devices

5.3.1.3 Pin description - NAND flashes

Pin	Driver (Device)
$\overline{CE}$	CHIP ENABLE The $\overline{CE}$ input enables the device. Signal is active low. If the signal is inactive, device is in standby mode.
$\overline{WE}$	WRITE ENABLE The $\overline{WE}$ input controls writes to the I/O port. Commands, address and data are latched on the rising edge of the WE pulse.
$\overline{RE}$	READ ENABLE The $\overline{RE}$ input is the serial data-out control. When active (low) the device outputs data.
CLE	COMMAND LATCH ENABLE This pin should be low, when writing commands to the command register.
ALE	ADDRESS LATCH ENABLE When active, an address can be written.
$\overline{WP}$	WRITE PROTECT Typically connected to VCC (recommended), but may also be connected to port pin.
$R/\overline{B}$	READY/BUSY OUTPUT The $R/\overline{B}$ output indicates the status of the device operation. When low, it indicates that a program, erase or read operation is in process. It returns to high state when the operation is completed. It is an open drain output. Should be connected to a port pin with pull-up. If available a port pin which can trigger an interrupt should be used.
I/O_0- I/O_7	DATA INPUTS/OUTPUTS The I/O pins are used to input command, address and data, and to output data during read operations.
I/O_8- I/O_{15}	DATA INPUTS/OUTPUTS I/O_8-I/O_{15} 16-bit flashes only.

Table 5.5: NAND flash pin description

5.3.1.4 Pin description - DataFlashes

DataFlash chips are commonly used when low pin count and easy data transfer are required. DataFlash devices use the following pins:

Pin	Meaning
CS	ChipSelect This pin selects the DataFlash device. The device is selected, when CS pin is driven low.
SCLK	Serial Clock The SCLK pin is an input-only pin and is used to control the flow of data to and from the DataFlash. Data is always clocked into the device on the rising edge of SCLK and clocked out of the device on the falling edge of SCLK.
SI	Serial Data In The SI pin is an input-only pin and is used to transfer data into the device. The SI pin is used for all data input including opcodes and address sequences.
SO	Serial Data Out This SO pin is an output pin and is used to transfer data serially out of the device.

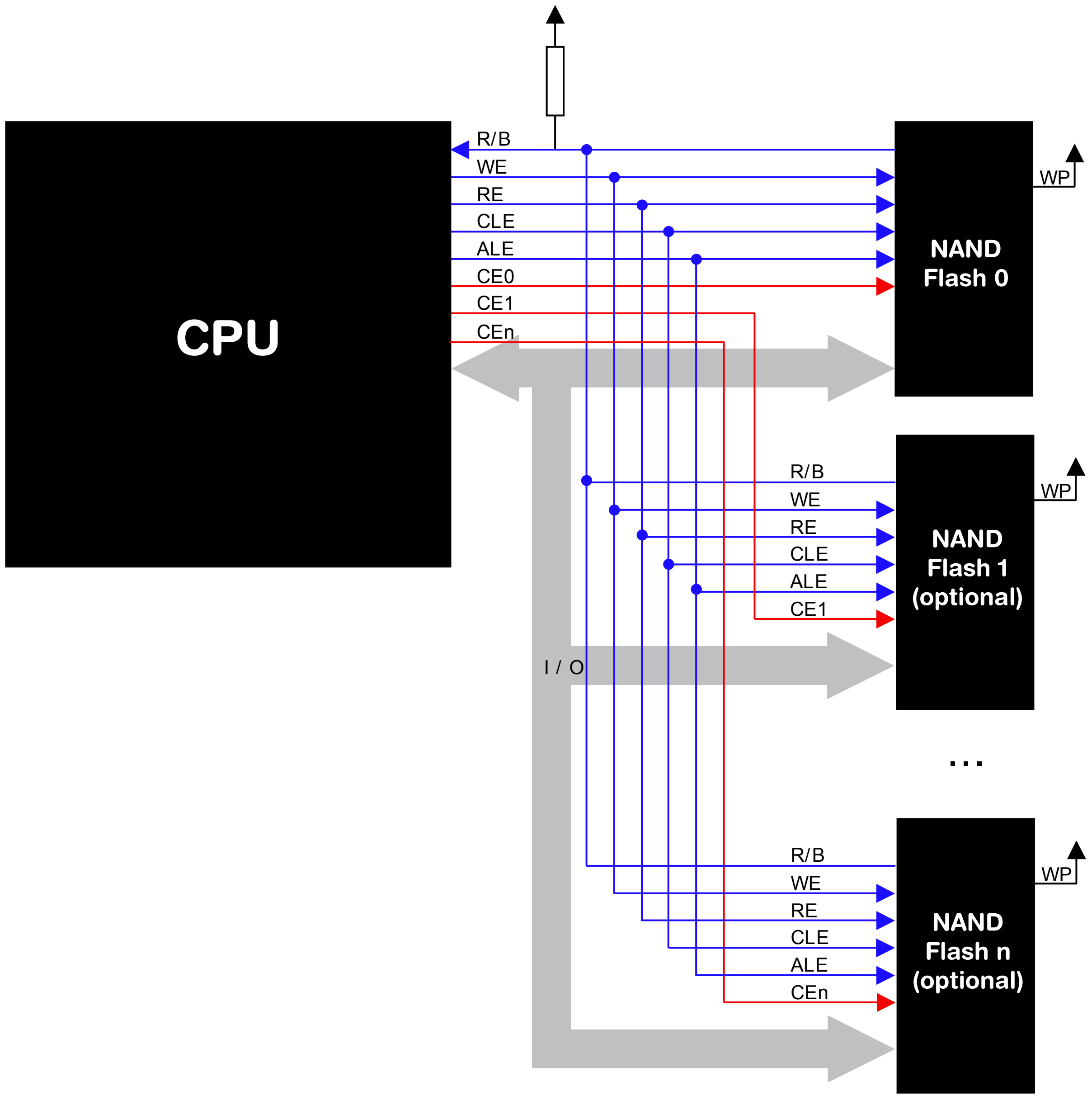
Table 5.6: DataFlash chip pin function description

Additionally the following requirements need to be fulfilled by your host system:

- Data transfer width is 8 bit.
- Chip Select (CS) sets the card active at low-level and inactive at high level.
- Clock signal must be generated by the target system. The serial flash chips are always in slave mode.
- Bit order requires most significant bit (MSB) to be sent out first.

To setup all these requirements, the NAND flash driver will call the function `FS_DF_HW_X_Init()`, therefore the function `FS_DF_HW_X_Init()` can be used to initialize the SPI bus. Refer to “FS_DF_HW_X_Init()” on page 183 for further details.

5.3.1.5 Sample block schematics



5.3.2 Theory of operation

NAND flash devices are divided into physical blocks and physical pages. One physical block is the smallest erasable unit; one physical page is the smallest writable unit. Small block NAND flashes contain multiple pages. One block contain typically 16 / 32 / 64 pages per block. Every page has a size of 528 bytes (512 data bytes + 16 spare bytes). Large block NAND Flash devices contain blocks made up of 64 pages, each page containing 2112 bytes (2048 data bytes + 64 spare bytes).

The driver uses the spare bytes for the following purposes:

1. To check if the data status byte and block status are valid.
   If they are valid the driver uses this sector. When the driver detects a bad sector, the

whole block is marked as invalid and its content is copied to a non-defective block.

2. To store/read an ECC (Error Correction Code) for data reliability.
   When reading a sector, the driver also reads the ECC stored in the spare area of the sector, calculates the ECC based on the read data and compares the ECCs. If the ECCs are not identical, the driver tries to recover the data, based on the read ECC.
   When writing to a page the ECC is calculated based on the data the driver has to write to the page. The calculated ECC is then stored in the spare area.

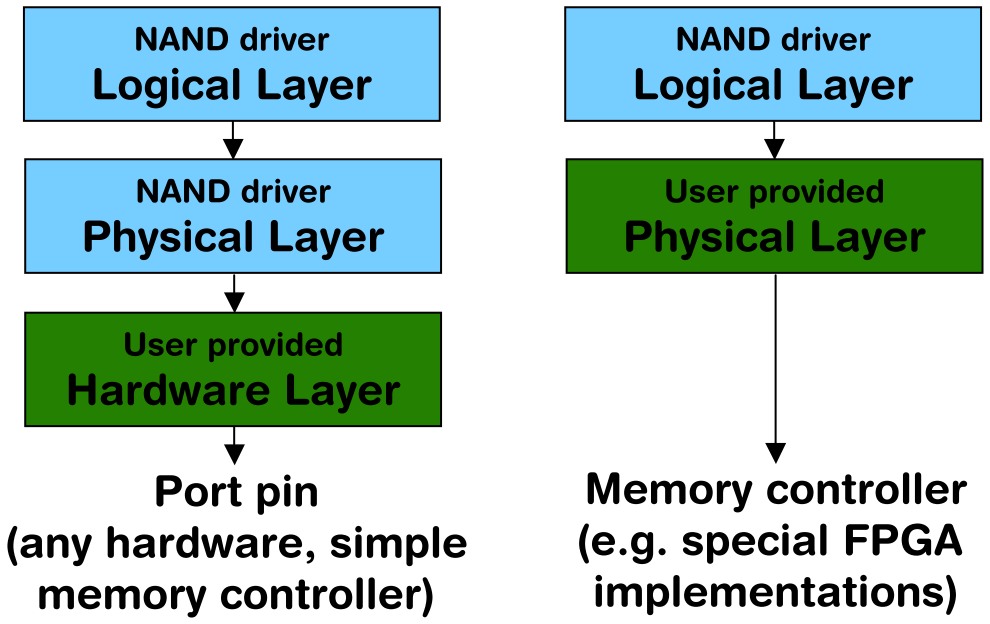
5.3.2.1 Error correction code (ECC)

The µC/FS NAND driver is highly speed optimized and offers a better error detection and correction than a standard memory controller ECC. The ECC is capable of single bit error correction and 2-bit random detection. When a block for which the ECC is computed has 2 or more bit errors, the data cannot be corrected.
Standard memory controllers compute an ECC for the complete blocksize (512 / 2048 bytes). The µC/FS NAND driver computes the ECC for data chunks of 256 bytes (e.g. a page with 2048 bytes is divided into 8 parts of 256 bytes), so the probability to detect and also correct data errors is much higher. This enhancement is realized with a very good performance. The ECC computation of the µC/FS NAND driver is highly optimized, so that a performance of 18 Mbytes/second can be achieved with a ARM7 running at 48 MHz.

We suggest the use of the the µC/FS NAND driver without the usage of a memory controller, because the performance of the driver is very high and the error correction is much better if it is controlled from driver side.

5.3.2.2 Software structure

The NAND Flash driver is split up into different layers, which are shown in the illustration below.


NAND driver Logical Layer
NAND driver Physical Layer
User provided Hardware Layer
Port pin (any hardware, simple memory controller)
NAND driver Logical Layer
User provided Physical Layer
Memory controller (e.g. special FPGA implementations)

It is possible to use the NAND driver with custom hardware. If port pins or a simple memory controller are used for accessing the flash memory, only the hardware layer needs to be ported, normally no changes to the physical layer are required. If the NAND driver should be used with a special memory controller (for example special FPGA implementations), the physical layer needs to be adapted. In this case, the hardware layer is not required, because the memory controller manages the hardware access.

5.3.3 Fail-safe operation

The µC/FS NAND driver is fail-safe. That means that the driver makes only atomic actions and takes the responsibility that the data managed by the file system is always valid. In case of a power loss or a power reset during a write operation, it is always assured that only valid data is stored in the flash. If the power loss interrupts the write operation, the old data will be kept and the block not corrupted.

5.3.4 Wear leveling

Wear leveling is supported by the driver. Wear leveling makes sure that the number of erase cycles remains approximately equal for each sector. Maximum erase count difference is set to 5. This value specifies a maximum difference of erase counts for different physical sectors before the wear leveling uses the sector with the lowest erase count.

5.3.5 Configuring the driver

5.3.5.1 Adding the driver to µC/FS

To add the driver, use `FS_AddDevice()` with the driver label `FS_NAND_Driver`. This function has to be called from within `FS_X_AddDevices()`. Refer to "FS_X_AddDevices()" on page 262 for more information.

Example

```
void FS_X_AddDevices(void) {
  FS_AddDevice(&FS_NAND_Driver);
  FS_NAND_SetPhyType(0, &FS_NAND_PHY_x8); // Set the physical
                                          // interface of the NAND flash
  FS_NAND_SetBlockRange(Unit, 2, 128);    // Skip 2 blocks (256 Kbytes in case of 2K
                                          // device)
                                          // Size is 128 blocks
                                          // For 2k devices, this means
                                          // 2 Kbytes * 64 * 128 = 16 Mbytes
}
```

5.3.5.2 Driver specific configuration functions

Function	Description
`FS_NAND_SetPhyType()`	Sets the physical type of the device.
`FS_NAND_SetBlockRange()`	Set a limit on which blocks of the NAND flash can be controlled by the driver.

Table 5.7: NAND driver specific configuration functions

FS_NAND_SetPhyType()

Description

Sets the physical type of the device. NAND flash is organized in pages of either 512 or 2048 bytes and has an 8-bit or 16-bit interface. The driver needs to know the correct combination of page and interface width.

Prototype

```
void FS_NAND_SetPhyType(U8 Unit, const FS_NAND_PHY_TYPE * pPhyType);
```

Parameter	Meaning
`Unit`	Unit number (0...N).
`pPhyType`	Pointer to physical type.

Table 5.8: FS_NAND_SetPhyType() parameter list

Permitted values for parameter `pPhyType`	
`FS_NAND_PHY_512x8`	Supports the following NAND devices: - 512 bytes per page and 8-bit width
`FS_NAND_PHY_2048x8`	Supports the following NAND devices: - 2048 bytes per page and 8-bit width
`FS_NAND_PHY_2048x16`	Supports the following NAND devices: - 2048 bytes per page and 16-bit width

Permitted values for parameter pPhyType	
FS_NAND_PHY_x	Supports the following NAND devices: - 512 bytes per page and 8-bit width - 2048 bytes per page and 8-bit width - 2048 bytes per page and 16-bit width
FS_NAND_PHY_x8	Supports the following NAND devices: - 512 bytes per page and 8-bit width - 2048 bytes per page and 8-bit width
FS_NAND_PHY_DataFlash	Supports ATMEL DataFlashes. The physical layer driver accesses these chips using the SPI mode. To use the driver with ATMEL DataFlash chips in your system, you will have to provide basic I/O functions which are divergend to the hardware functions of the other physical layers. Refer to ...

Additional information

This function needs to be called for every NAND device added.

Example

Refer to “Adding the driver to µC/FS” on page 155 for an example.

FS_NAND_SetBlockRange()

Description

Sets a limit for which blocks of the NAND flash can be controlled by the driver.

Prototype

```
void FS_NAND_SetBlockRange(U8 Unit, U16 FirstBlock, U16 MaxNumBlocks);
```

Parameter	Meaning
Unit	Unit number (0...N).
FirstBlock	Zero-based index of the first block to use. Specifies the number of blocks at the beginning of the device to skip. 0 means that no blocks are skipped.
MaxNumBlocks	Maximum number of blocks to use. 0 means use all blocks after FirstBlock.

Table 5.9: FS_NAND_SetBlockRange() parameter list

Additional information

This function is optional. By default, the driver controls all blocks of the NAND flash, making the entire NAND flash available. If a part of the NAND flash should be used for another purpose (for example to store the application program used by a bootloader) and therefore is not controlled by the driver, this function can be used. Limiting the number of blocks used by the driver also reduces the amount of memory used by the driver.

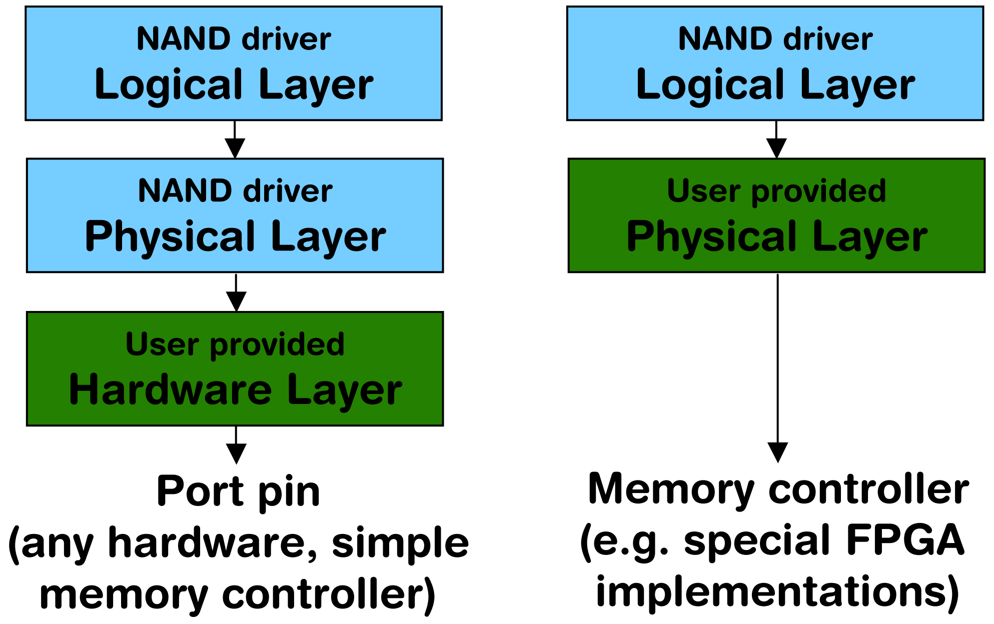
Example

Refer to “Adding the driver to µC/FS” on page 155 for an example.

5.3.6 Hardware functions: Physical layer functions

There is normally no need to change the physical layer of the NAND driver, only the hardware layer has to be adapted.

In some special cases, when the low-level hardware routines provided by the driver are not compatible with the target hardware (e.g. special FPGA implementations of a memory controller), the physical layer has to be adapted.



If there is a reason to change the physical layer anyhow, the functions which have to be changed are organized in a function table. The function table is implemented in a structure of type `FS_NAND_PHY_TYPE`.

```
typedef struct FS_NAND_PHY_TYPE {
  int     (*pfEraseBlock)        (U8                     Unit,
                                  U32                    Block);
  int     (*pfInitGetDeviceInfo) (U8                     Unit,
                                  FS_NAND_DEVICE_INFO *  pDevInfo);
  int     (*pfIsWP)              (U8                     Unit);
  int     (*pfRead)              (U8                     Unit,
                                  U32                    PageNo,
                                  void *                 pData,
                                  unsigned               Off,
                                  unsigned               NumBytes);
  int     (*pfReadEx)            (U8                     Unit,
                                  U32                    PageNo,
                                  void *                 pData,
                                  unsigned               Off,
                                  unsigned               NumBytes,
                                  void *                 pSpare,
                                  unsigned               OffSpare,
                                  unsigned               NumBytesSpare);
  int     (*pfWrite)             (U8                     Unit,
                                  U32                    PageNo,
                                  const void *           pData,
                                  unsigned               Off,
                                        unsigned               NumBytes);
  int    (*pfWriteEx)                  (U8                     Unit,
                                        U32                    PageNo,
                                        const void *           pData,
                                        unsigned               Off,
                                        unsigned               NumBytes,
                                        const void *           pSpare,
                                        unsigned               OffSpare,
                                        unsigned               NumBytesSpare);
} FS_NAND_PHY_TYPE;
```

If the physical layer should be modified, the following members of the structure `FS_NAND_PHY_TYPE` have to be adapted:

Routine	Explanation
`*pfEraseBlock`	Erases a chosen block of the device.
`*pfInitGetDevInfo()`	Initializes the devices and retrieves the device information.
`*pfIsWP`	Checks if the device is write protected.
`*pfRead`	Reads data from the device.
`*pfReadEx`	Reads data from the device and the spare area.
`*pfWrite`	Writes data to the device.
`*pfWriteEx`	Writes data to the device and the spare area.

Table 5.10: NAND device driver physical layer functions

5.3.6.1 (*pfEraseBlock)()

Description

Erases one block of the device. A block is the smallest eraseable unit.

Prototype

```
int (*pfEraseBlock) (U8 Unit, U32 PageIndex);
```

Parameter	Meaning
`Unit`	Unit number (0...N).
`PageIndex`	Zero-based index of the first page in the block to be erased. If the device has 64 pages per block, then the following values are permitted: `PageIndex` == 0 -> block 0, `PageIndex` == 64 -> block 1, `PageIndex` == 128 -> block 2, etc.

Table 5.11: (*pfEraseBlock)() parameter list

Return value

== 0: On success, block erased.
==-1: In case of an error.

5.3.6.2 (*pfInitGetDeviceInfo)()

Description

Initializes hardware layer, resets NAND flash and tries to identify the NAND flash. If the NAND flash can be handled, `FS_NAND_DEVICE_INFO` is filled.

Prototype

```
int (*pfInitGetDeviceInfo) (U8                    Unit,
                            FS_NAND_DEVICE_INFO * pDevInfo) {
```

Parameter	Meaning
`Unit`	Unit number (0...N).
`pDevInfo`	Pointer to a structure of type `FS_NAND_DEVICE_INFO`.

Table 5.12: (*pfInitGetDeviceInfo)() parameter list

Return value

== 0: On success.
== 1: In case of an error.

5.3.6.3 (*pfIsWP)()

Description

Checks if the device is write protected. This is done by reading bit 7 of the status register. Typical reason for write protection is that either the supply voltage is too low or the /WP-pin is active (low).

Prototype

```
int (*pfIsWP)(U8 Unit);
```

Parameter	Meaning
`Unit`	Unit number (0...N).

Table 5.13: (*pfIsWP)() parameter list

Return value

== 0: Device is not write protected.
== 1: Device is write protected.

5.3.6.4 (*pfRead)()

Description

This function can be used to read from the data or spare area of the device. The spare area is assumed to be located right after the main area.

Prototype

```
int (*pfRead) (U8         Unit,
               U32        PageIndex,
               void *     pData,
               unsigned   Off,
               unsigned   NumBytes);
```

Parameter	Meaning
Unit	Unit number (0...N).
PageIndex	Zero-based index of page to be read. Needs to be smaller than page size.
pData	Pointer to a buffer for read data.
Off	Byte offset within the page.
NumBytes	Number of bytes to read

Table 5.14: (*pfRead)() parameter list

Return value

== 0: Data successfully transferred.
!= 0: An error has occurred.

Additional information

If the parameter Off is smaller than the page size, the data area is accessed. An offset greater than the page size indicates that the spare area should be accessed.

5.3.6.5 (*pfReadEx)()

Description

Reads from both the data and the spare area of a page.

Prototype

```
int     (*pfReadEx) (U8                     Unit,
                     U32                    PageIndex,
                     void *                 pData,
                     unsigned               Off,
                     unsigned               NumBytes,
                     void *                 pSpare,
                     unsigned               OffSpare,
                     unsigned               NumBytesSpare);
```

Parameter	Meaning
`Unit`	Unit number (0...N).
`PageIndex`	Number of page that should be read.
`pData`	Pointer to a buffer for read data.
`Off`	Byte offset within the page, which needs to be smaller than the page size.
`NumBytes`	Number of bytes to read.
`pSpare`	Pointer to a buffer for spare data.
`OffSpare`	Offset from the start of the spare area to the point where spare data should be read. First byte of the spare area has the same off-set as the page size. Example: Page size: 512 OffSpare == 512 -> First byte of spare area OffSpare == 513 -> Second byte of spare area Page size: 2048 OffSpare == 2048 -> First byte of spare area OffSpare == 2049 -> Second byte of spare area
`NumBytesSpare`	Number of spare bytes to read.

Table 5.15: (*pfReadEx)() parameter list

Return value

== 0: Data successfully transferred.
!= 0: An error has occurred.

5.3.6.6 (*pfWrite)()

Description

Writes data into a complete or a part of a page. This code is identical for main memory and spare area; the spare area is located rigth after the main area.

Prototype

```
int (*pfWrite) (U8             Unit,
                U32            PageIndex,
                const void *   pData,
                unsigned       Off,
                unsigned       NumBytes);
```

Parameter	Meaning
Unit	Unit number (0...N).
PageIndex	Zero-based index of page to be written.
pData	Pointer to a buffer of data which should be written.
Off	Byte offset within the page, which needs to be smaller than the page size.
NumBytes	Number of bytes which should be written.

Table 5.16: (*pfWrite)() parameter list

Return value

== 0: Data successfully transferred.
!= 0: An error has occurred.

5.3.6.7 (*pfWriteEx)()

Description

Writes data to 2 parts of a page. Typically used to write both the data and spare area of a page in one step.

Prototype

```
int (*pfWriteEx) (U8                     Unit,
                  U32                    PageIndex,
                  const void *           pData,
                  unsigned               Off,
                  unsigned               NumBytes,
                  const void *           pSpare,
                  unsigned               OffSpare,
                  unsigned               NumBytesSpare);
```

Parameter	Meaning
Unit	Unit number (0...N).
PageIndex	Number of page that should be written.
pData	Pointer to a buffer of data which should be written.
Off	Byte offset within the page, which needs to be smaller than the page size.
NumBytes	Number of bytes to write.
pSpare	Pointer to a buffer data which should be written to the spare area.
OffSpare	Offset from the start of the spare area to the point where spare data should be written. First byte of the spare area has the same offset as the page size. Example: Page size: 512 OffSpare == 512 -> First byte of spare area OffSpare == 513 -> Second byte of spare area Page size: 2048 OffSpare == 2048 -> First byte of spare area OffSpare == 2049 -> Second byte of spare area
NumBytesSpare	Number of spare bytes to write.

Table 5.17: (*pfWriteEx)() parameter list

Return value

== 0: Data successfully transferred.
!= 0: An error has occurred.

5.3.7 Hardware functions

Routine	Explanation
FS_NAND_HW_X_SetAddrMode()	CLE low and ALE high for the specified device.
FS_NAND_HW_X_SetCmdMode()	CLE high and ALE low for the specified device.
FS_NAND_HW_X_SetDataMode()	CLE low and ALE low for the specified device.
FS_NAND_HW_X_DisableCE()	Disables CE.
FS_NAND_HW_X_EnableCE()	Enables CE.
FS_NAND_HW_X_WaitWhileBusy()	Waits while the device is busy.
FS_NAND_HW_X_Read_x8()	For 8-bit NAND flashes: Reads data from the NAND flash device.
FS_NAND_HW_X_Read_x16()	For 16-bit NAND flashes: Reads data from the NAND flash device.
FS_NAND_HW_X_Write_x8()	For 8-bit NAND flashes: Writing data to the NAND flash, using the I/O 0-7 lines of the NAND flash device.
FS_NAND_HW_X_Write_16()	For 16-bit NAND flashes: Writing data to the NAND flash, using the I/O 0-15 lines of the NAND flash device.
FS_NAND_HW_X_Delayus()	Delays for a specified period of time.
FS_NAND_HW_X_Init_x8()	For 8-bit NAND flashes: Initializes the NAND flash device.
FS_NAND_HW_X_Init_x16()	For 16-bit NAND flashes: Initializes the NAND flash device.

Table 5.18: NAND device driver hardware layer functions

5.3.7.1 FS_NAND_HW_X_SetAddrMode()

Description

Sets CLE low and ALE high for the specified device.

Prototype

```
void FS_NAND_HW_X_SetAddrMode (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0...N).

Table 5.19: FS_NAND_HW_X_SetAddrMode() parameter list

Additional Information

This function is called to start the address data transfer.

Example

```
void FS_NAND_HW_X_SetAddrMode(U8 Unit) {
  FS_USE_PARA(Unit);
  /* CLE low, ALE high */
  NAND_CLR_CLE();
  NAND_SET_ALE();}
}
```

5.3.7.2 FS_NAND_HW_X_SetCmdMode()

Description

Sets CLE high and ALE low for the specified device.

Prototype

```
void FS_NAND_HW_X_SetCmdMode (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0...N).

Table 5.20: FS_NAND_HW_X_SetCmdMode() parameter list

Additional Information

This function is called to start the command transfer.

Example

```
void FS_NAND_HW_X_SetCmdMode(U8 Unit) {
  FS_USE_PARA(Unit);
  /* CLE high, ALE low  */
  NAND_SET_CLE();
  NAND_CLR_ALE();
}
```

5.3.7.3 FS_NAND_HW_X_SetDataMode()

Description

Sets CLE low and ALE low for the specified device.

Prototype

```
void FS_NAND_HW_X_SetDataMode (U8 Unit);
```

Parameter	Meaning
`Unit`	Unit number (0...N).

Table 5.21: FS_NAND_HW_X_SetDataMode() parameter list

Additional Information

This function is called to the start data transfer.

Example

```
void FS_NAND_HW_X_SetData(U8 Unit) {
  FS_USE_PARA(Unit);
  /* CLE low, ALE low  */
  NAND_CLR_CLE();
  NAND_CLR_ALE();
}
```

5.3.7.4 FS_NAND_HW_X_DisableCE()

Description

Disables NAND CE.

Prototype

```
void FS_NAND_HW_X_DisableCE (U8 Unit);
```

Parameter	Description
`Unit`	Unit number (0...N).

Table 5.22: FS_NAND_HW_X_DisableCE() parameter list

5.3.7.5 FS_NAND_HW_X_EnableCE()

Description

Enables NAND CE.

Prototype

```
void FS_NAND_HW_X_EnableCE (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0...N).

Table 5.23: FS_NAND_HW_X_EnableCE() parameter list

Example

```
/*********************************************************************
*
*       FS_NAND_HW_X_EnableCE
*/
void FS_NAND_HW_X_EnableCE(U8 Unit) {
  PIOB_CODR = (1 << 18); // Enable NAND CE
}
```

5.3.7.6 FS_NAND_HW_X_WaitWhileBusy()

Description

Checks whether the device is busy.

Prototype

```
int FS_NAND_HW_X_WaitWhileBusy (U8       Unit,
                                unsigned us);
```

Parameter	Meaning
Unit	Unit number (0...N).
us	Time in µs to wait.

Table 5.24: FS_NAND_HW_X_WaitWhileBusy() parameter list

Return value

`0` if the device is not busy.
Any other value means that an operation is pending.

Additional Information

If your hardware allows you to monitor the nR/B line, you can use the status of that line and return when the device is not busy. Otherwise, the function should return 1. In this case, the physical layer will perform a software-status-check of the device or wait for the time required by the current operation.

Example

```
int FS_NAND_HW_X_WaitWhileBusy(U8 Unit, unsinged us) {
  int IsReady;
  do {
    IsReady = NAND_GET_RDY() ? 0 : 1;
  } while(IsReady == 0);
  return IsReady;
}
```

5.3.7.7 FS_NAND_HW_X_Read_x8()

Description

Reads data from an 8-bit NAND flash device, using the I/O 0-7 lines.

Prototype

```
void FS_NAND_HW_X_Read_x8 (U8        Unit,
                           U8 *      pBuffer,
                           unsigned  NumBytes);
```

Parameter	Meaning
Unit	Unit number (0...N).
pBuffer	Pointer to a buffer to store the read data.
NumBytes	Number of bytes that should be stored into the buffer.

Table 5.25: FS_NAND_HW_X_Read_x8() parameter list

Additional Information

When reading from the device, usually you will not have to take care of handling the RE line because that is done automatically by the hardware.
If you do have to control the RE line, make sure that timing is according to your NAND flash device specification.

Example

```
void FS_NAND_HW_X_Read_x8(U8 Unit, U8 * pBuffer, unsigned NumBytes) {
  SET_DATA2INPUT();
  do {
    NAND_CLR_RE();      /*  RE is active low */
    NAND_GET_DATA(*pBuffer);
    pBuffer++;
    NAND_SET_RE();     /* disable RE */
  } while (--NumBytes);
}
```

5.3.7.8 FS_NAND_HW_X_Read_x16()

Description

Reads data from a 16-bit NAND flash device, using the I/O 0-15 lines.

Prototype

```
void FS_NAND_HW_X_Read_x16 (U8        Unit,
                            U8 *      pBuffer,
                            unsigned  NumBytes);
```

Parameter	Meaning
Unit	Unit number (0...N).
pBuffer	Pointer to a buffer to store the read data.
NumBytes	Number of bytes that should be stored into the buffer.

Table 5.26: FS_NAND_HW_X_Read_x16() parameter list

Additional Information

When reading from the device, usually you will not have to take care of handling the RE line because that is done automatically by the hardware.
If you do have to control the RE line, make sure that timing is according to your NAND flash device specification.

5.3.7.9 FS_NAND_HW_X_Write_x8()

Description

Writes data to an 8-bit NAND flash, using the I/O 0-7 lines of the NAND flash device.

Prototype

```
void FS_NAND_HW_X_Write_x8 (U8          Unit,
                            const U8 *  pBuffer,
                            unsigned    NumBytes);;
```

Parameter	Meaning
Unit	Unit number (0...N).
pBuffer	Pointer to a buffer of data which should be written.
NumBytes	Number of bytes that should be transferred to the NAND flash.

Table 5.27: FS_NAND_HW_X_Write_x8() parameter list

Additional Information

When writing data to the device, usually you will not have to take care of handling the WE line because that is done automatically by the hardware.
If you do have to control the WE line, make sure that timing is according to your NAND flash device specifications.

Example

```
void FS_NAND_HW_X_Write_x8(U8 Unit, U8 * pBuffer, unsigned NumBytes) {
  SET_DATA2OUTPUT();
  do {
    NAND_CLR_WE();     /*  WE is active low */
    NAND_SET_DATA(*pBuffer);
    pBuffer++;
    NAND_SET_WE();    /* disable WE */
  } while (--NumBytes);
 }
```

5.3.7.10 FS_NAND_HW_X_Write_x16()

Description

Writing data to a 16-bit NAND flash, using the I/O 0-15 lines of the NAND flash device.

Prototype

```
void FS_NAND_HW_X_Write_x16 (U8          Unit,
                             const U8 *  pBuffer,
                             unsigned    NumBytes);;
```

Parameter	Meaning
`Unit`	Unit number (0...N).
`pBuffer`	Pointer to a buffer of data which should be written.
`NumBytes`	Number of bytes that should be transferred to the NAND flash.

Table 5.28: FS_NAND_HW_X_Write_x16() parameter list

Additional Information

When writing data to the device, usually you will not have to take care of handling the WE line because that is done automatically by the hardware.
If you do have to control the WE line, make sure that timing is according to your NAND flash device specifications.

5.3.7.11 FS_NAND_HW_X_Delayus()

Description

Delays for a specified period of time.

Prototype

```
void FS_NAND_HW_X_Delayus (unsigned us);
```

Parameter	Meaning
`us`	Time to dwell in µs.

Table 5.29: FS_NAND_HW_X_Delayus() parameter list

Additional Information

The dwell time is specified in µs. It is the user's responsibility to make sure that this function will at least wait the time specified by `us`.
The driver uses this function in situations where a minimum delay time is required by the specification of the NAND flash.
It is called only if the busy pin of the NAND flash cannot be read (`FS_NAND_HAS_BUSY_PIN` == 0)

Example C

```
void FS_NAND_HW_X_Delayus(unsigned us) {
  volatile int i;
  i = 5 * us;    // Factor depends on CPU speed
  while (i--);
}
```

Example ARM assembly

```
FS_NAND_HW_X_Delayus:
        mov         R1, #10   ; Factor depends on CPU speed
        mul         R0, R0, R1
        Loop:
        subs        R0, R0, #1
        bne         Loop
        BX          LR                  ;; return
```

5.3.7.12 FS_NAND_HW_X_Init_x8()

Description

Initializes a NAND flash device with an 8-bit interface.

Prototype

```
void FS_NAND_HW_X_Init_x8 (U8 Unit);
```

Parameter	Meaning
`Unit`	Unit number (0...N).

Table 5.30: FS_NAND_HW_X_Init_x8() parameter list

Additional Information

This function is called before any access to the NAND flash device is made. Use this function to initialize the hardware.

Example

```
int FS_NAND_HW_X_Init_x8(U8 Unit) {
 FS_USE_PARA(Unit);
 _Timer2Config();
 _NANDFlashInit();
}
```

5.3.7.13 FS_NAND_HW_X_Init_x16()

Description

Initializes a NAND flash device with a 16-bit interface.

Prototype

```
void FS_NAND_HW_X_Init_x16 (U8 Unit);
```

Parameter	Meaning
`Unit`	Unit number (0...N).

Table 5.31: FS_NAND_HW_X_Init_x16() parameter list

Additional Information

This function is called before any access to the NAND flash device is made. Use this function to initialize the hardware.

5.3.8 Hardware functions - ATMEL DataFlash

Routine	Explanation
Control line functions	
`FS_DF_HW_X_EnableCS()`	Activates chip select signal (CS) of the DataFlash chip.
`FS_DF_HW_X_DisableCS()`	Deactivates chip select signal (CS) of the DataFlash chip.
`FS_DF_HW_X_Init()`	Initializes the SPI hardware.
Data transfer functions	
`FS_DF_HW_X_Read()`	Receives a number of bytes from the DataFlash.
`FS_DF_HW_X_Write()`	Sends a number of bytes to the card.

Table 5.32: DataFlash device driver hardware functions

5.3.8.1 FS_DF_HW_X_EnableCS()

Description

Activates chip select signal (CS) of the specified DataFlash.

Prototype

```
void FS_DF_HW_X_EnableCS (U8 Unit);
```

Parameter	Meaning
`Unit`	Unit number (0...N).

Table 5.33: FS_DF_HW_X_EnableCS() parameter list

Additional Information

The CS signal is used to address a specific DataFlash chip connected to the SPI. Enabling is equal to setting the CS line to low.

Example

```
void FS_DF_HW_X_EnableCS(U8 Unit) {
  SPI_CLR_CS();
}
```

5.3.8.2 FS_DF_HW_X_DisableCS()

Description

Deactivates chip select signal (CS) of the specified DataFlash.

Prototype

```
void FS_DF_HW_X_DisableCS (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0...N).

Table 5.34: FS_DF_HW_X_DisableCS() parameter list

Additional Information

The CS signal is used to address a specific DataFlash connected to the SPI. Disabling is equal to setting the CS line to high.

Example

```
void FS_DF_HW_X_DisableCS(U8 Unit) {
  SPI_SET_CS();
}
```

5.3.8.3 FS_DF_HW_X_Init()

Description

Initializes the SPI hardware.

Prototype

```
int FS_DF_HW_X_Init (U8 Unit);
```

Parameter	Meaning
`Unit`	Unit number (0...N).

Table 5.35: FS_DF_HW_X_Init() parameter list

Return value

== 0 Initialization was successful.
== 1 Initialization failed.

Additional Information

The `FS_DF_HW_X_Init()` can be used to initialize the SPI hardware. As described in the previous section. The SPI should be initialized as follows:

- 8-bit data length
- MSB should be sent out first
- CS signal should be initially high
- The set clock frequency should not exceed the max clock frequency that are specified by the Serial Flash devices (Usually: 20MHz).

Example

```
void FS_DF_HW_X_Init(U8 Unit) {
  SPI_SETUP_PINS();
}
```

5.3.8.4 FS_DF_HW_X_Read()

Description

Receives a number of bytes from the DataFlash.

Prototype

```
void FS_DF_HW_X_Read (U8   Unit,
                      U8 * pData,
                      int  NumBytes);
```

Parameter	Meaning
Unit	Unit number (0...N).
pData	Pointer to a buffer for data to be receive.
NumBytes	Number of bytes to receive.

Table 5.36: FS_DF_HW_X_Read() parameter list

Additional Information

This function is used to read a number of bytes from a card to buffer memory.

Example

```
void FS_DF_HW_X_Read (U8 Unit, U8 * pData, int NumBytes) {
  do {
    c = 0;
    bpos = 8; /* get 8 bits */
    do {
      SPI_CLR_CLK();
      c <<= 1;
      if (SPI_DATAIN()) {
        c |= 1;
      }
      SPI_SET_CLK();
    } while (--bpos);
    *pData++ = c;
  } while (--NumBytes);
}
```

5.3.8.5 FS_DF_HW_X_Write()

Description

Sends a number of bytes to the card.

Prototype

```
void FS_DF_HW_X_Write (U8          Unit,
                       const U8 *  pData,
                       int         NumBytes);
```

Parameter	Meaning
Unit	Unit number (0...N).
pData	Pointer to a buffer for data to be receive.
NumBytes	Number of bytes to be written.

Table 5.37: FS_DF_HW_X_Write() parameter list

Additional Information

This function is used to send a number of bytes from memory buffer to the dedicated DataFlash.

Example

```
void FS_DF_HW_X_Write(U8 Unit, const U8 * pData, int NumBytes) {
  int i;
  U8 mask;
  U8 data;
  for (i = 0; i < NumBytes; i++) {
    data = pData[i];
    mask = 0x80;
    while (mask) {
      if (data & mask) {
        SPI_SET_DATAOUT();
      } else {
        SPI_CLR_DATAOUT();
      }
      SPI_CLR_CLK();
      SPI_DELAY();
      SPI_SET_CLK();
      SPI_DELAY();
      mask >>= 1;
    }
  }
  SPI_SET_DATAOUT(); /* default state of data line is high */
}
```

5.3.9 Additional Information

Low-level format

Before using the NAND flash as a storage device, a low-level format has to be performed. Refer to "FS_FormatLow()" on page 83 and "FS_FormatLLIfRequired()" on page 82 for detailed information about low-level format.

5.3.10 Resource usage

5.3.10.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the NAND driver presented in the tables below have been measured on a system as follows: ARM7, IAR Embedded workbench V4.41A, Thumb mode, Size optimization.

Module	ROM [Kbytes]
µC/FS NAND driver	4.5

In addition, one of the following physical layers is required:

Physical layer	Description	ROM [Kbytes]
`FS_NAND_PHY_512x8`	Physical layer for small NAND devices with an 8-bit interface.	1.1
`FS_NAND_PHY_2048x8`	Physical layer for large NAND devices with an 8-bit interface.	1.0
`FS_NAND_PHY_2048x16`	Physical layer for large NAND devices with an 16-bit interface.	1.0
`FS_NAND_PHY_x8`	Physical layer for large and small NAND devices with an 8-bit interface.	2.3
`FS_NAND_PHY_x`	Physical layer for large and small NAND devices with an 8-bit or 16-bit interface.	3.3

5.3.10.2 Static RAM usage

Static RAM usage is the amount of RAM required by the driver for variables inside of the driver. The number of bytes can be seen in a list file.

Static RAM usage of the NAND driver: 32 bytes

5.3.10.3 Run-time RAM usage

Run-time RAM usage is the amount of RAM allocated by the driver at runtime. The amount required depends on the runtime configuration and the connected device.

The approximately RAM usage for the NAND driver can be calculated as follows:

Every NAND device requires:
`160 + 2 * NumberOfUsedBlocks + 4 * SectorsPerBlock + 1.04 * MaxSectorSize`

Example: 2 GBit NAND flash with 2K pages, 2048 blocks used, 512-byte sectors

One block consists of 64 pages, each page holds 4 sectors of 512 bytes.

SectorsPerBlock = 256
NumberOfUsedBlocks = 2048
MaxSectorSize = 512

RAM usage = (160 + 2 * 2048 + 4 * 256 + 1.04 * 512) bytes
RAM usage = 5813 bytes

Example: 2 GBit NAND flash with 2K pages, 2048 blocks used, 2048-byte sectors

One block consists of 64 pages, each page holds 1 sector of 2048 bytes.

SectorsPerBlock = 64
NumberOfUsedBlocks = 2048
MaxSectorSize = 2048

RAM usage = (160 + 2 * 2048 + 4 * 64 + 1.04 * 2048) bytes
RAM usage = 6642bytes

Example: 512 MBit NAND flash with 512 pages, 4096 blocks used, 512-byte sectors

One block consists of 64 pages, each page holds 1 sector of 512 bytes.

SectorsPerBlock = 32
NumberOfUsedBlocks = 8192
MaxSectorSize = 512

RAM usage = (160 + 2 * 4096 + 4 * 32 + 1.04 * 512) bytes
RAM usage = 9013 bytes

5.3.11 FAQs

Q: Are Multi-Level Cell NAND flashes (MLCs) supported ?
A: No, the driver does not support MLCs.

Q: Are NAND flashes with 4-Kbytes pages supported ?
A: Not yet, but this will be added. You should get in touch with us to find out what the current status is.

5.4 MultiMedia & SD card driver

µC/FS supports MultiMedia, SecureDigital (SD) and SecureDigital High Capacity (SDHC) cards. Two optional generic drivers for MultiMedia, SecureDigital (SD), SecureDigital High Capacity (SDHC), Mini SecureDigital and Micro SecureDigital cards are available.

MultiMedia & SecureDigital (SD) cards can be accessed though two different modes:

- SPI MODE
- MMC/SD card mode.

Drivers are available for both modes.
To use one of these drivers, you need to configure the selected MultiMedia & SD card driver and provide basic I/O functions for accessing your card reader hardware.

This section describes how to enable one of these drivers and all hardware access functions required by µC/FS's for either the MultiMedia, SD and SDHC card in SPI mode or MultiMedia & SD card mode driver.

5.4.1 Supported hardware

MultiMedia Cards (MMC), SecureDigital Cards (SD card), and SecureDigital High Capacity (SDHC) Cards are small-size factor mass storage devices.

The main design goal of these devices are to provide a very low cost mass storage product, implemented as a card with a simple controlling unit, and a compact, easy-to-implement interface. These requirements lead to a reduction of the functionality of each card to an absolute minimum. In order to have a flexible design, MMC, SD and SDHC cards are designed to be used in different I/O modes:

- MMC/SD card mode
- SPI mode
- Emulated SPI mode, using port pins.

The difference between MMC and SD cards are that SD cards can operate with a higher clock frequency. The clock range can be between 0 - 25MHz, whereas MMCs can only operate up to 20MHz. Additionally the initialization of these cards differs. They need to be initialized differently, but after initialization they behave the same way.

MMC and SD cards also differ in the count of pins. SD cards have more pins than MMCs. Depending in which mode they are used the pins are used or not. Additionally SD cards have a write protect switch, which can be used to lock the data on the card.

In MMC/SD card mode

MultiMedia Cards use a seven pin interface in MultiMedia card mode (Command, Clock, Data and 3x Power lines). In contrast to the MultiMedia cards, SD cards use a 9 pin interface (Command, Clock, 1 or 4 Data and 3 Power lines).

In SPI mode

Both card systems use the same pin interface. (ChipSelect (CS), DataIn, DataOut, Clock and 3 power lines).

5.4.1.1 Pin description for MMC/SD card in Card mode

Pin No.	Name	Type	Description
1	CD/ DAT[3]	Input/Output using push pull drivers	Card Detect / Data line [Bit 3] After power up this line is input with 50-kOhm pull-up resistor. This can be used for card detection; relevant only for SD cards. The pull-up resistor is disabled after the initialization procedure for using this line as DAT3, Data line[Bit 3], for data transfer.
2	CMD	Push Pull	Command/Response CMD is a bidirectional command channel used for card initialization and data transfer commands. The CMD signal has two operation modes: open-drain for initialization mode and push-pull for fast command transfer. Commands are sent from the MultiMediaCard bus master (card host controller) to the card and responses are sent from the cards to the host.
3	V_{SS}	Power supply	Supply voltage ground.
4	V_{DD}	Power supply	Supply voltage.
5	CLK	Input	Clock signal With each cycle of this signal an one bit transfer on the command and data lines is done. The frequency may vary between zero and the maximum clock frequency.
6	V_{SS2}	Power supply	Supply voltage ground.
7	DAT0	Input/Output using push pull drivers	Data line [Bit 0] DAT is a bidirectional data channel. The DAT signal operates in push-pull mode. Only one card or the host is driving this signal at a time. Relevant only for SD cards: For data transfers, this line is the Data line [Bit 0].
8	DAT1	Input/Output using push pull drivers	Data line [Bit 1] On MMC card this line does not exist. Relevant only for SD cards: For data transfer, this line is the Data line [Bit 1]. Connect an external pull-up resistor to this data line even if only DAT0 is to be used. Otherwise, non-expected high current consumption may occur due to the floating inputs of DAT1.
9	DAT2	Input/Output using push pull drivers	Data line [Bit 2] On MMC card this line does not exist. Relevant only for SD cards: For data transfer, this line is the Data line [Bit 2]. Connect an external pull-up resistor to this data line even if only DAT0 is to be used. Otherwise, non-expected high current consumption may occur due to the floating inputs of DAT2.

Table 5.38: MMC/SD card pin description

5.4.1.2 Pin description for MMC/SD card in SPI mode

Pin No.	Name	Type	Description
1	CS	Input	Chip Select It sets the card active at low-level and inactive at high level.
2	MOSI	Input	Master Out Slave In Transmits data to the card.
3	V_{SS}	Supply ground	Power supply ground Supply voltage ground.
4	V_{DD}	Supply voltage	Supply voltage.
5	SCLK	Input	Clock signal It must be generated by the target system. The card is always in slave mode.
6	V_{SS2}	Supply ground	Supply voltage ground.
7	MISO	Output	Master In Slave Out Line to transfer data to the host.
8	Reserved	Not used	The reserved pin is a floating input. Therefore, connect an external pull-up resistor to it. Otherwise, non-expected high current consumption may occur due to the floating input.
9	Reserved	Not used	The reserved pin is a floating input. Therefore, connect an external pull-up resistor to it. Otherwise, non-expected high current consumption may occur due to the floating input.

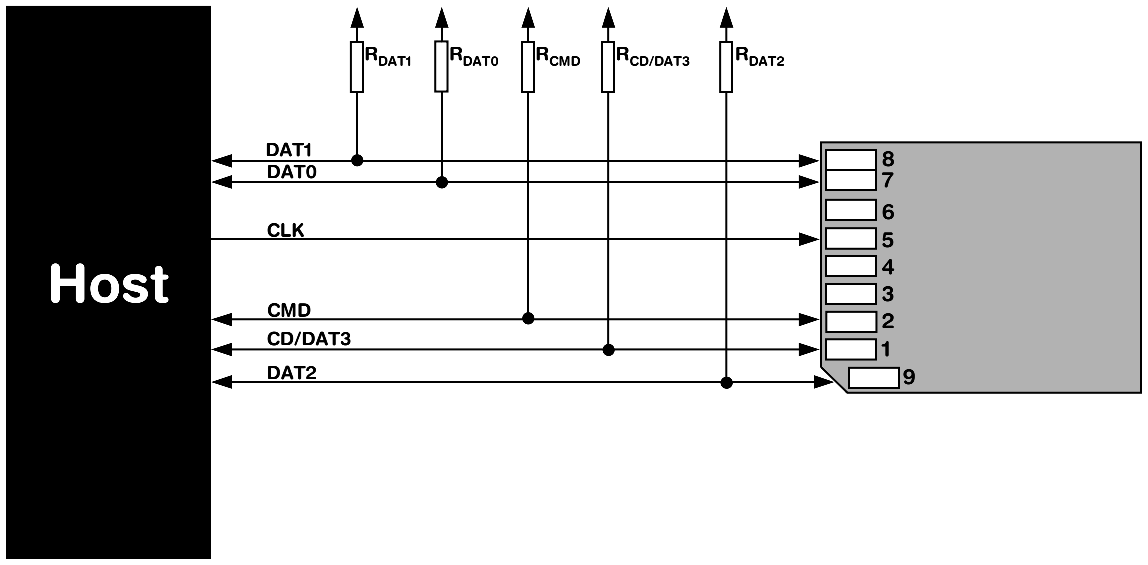
Table 5.39: MMC/SD card (SPI mode) pin description

Additional Information

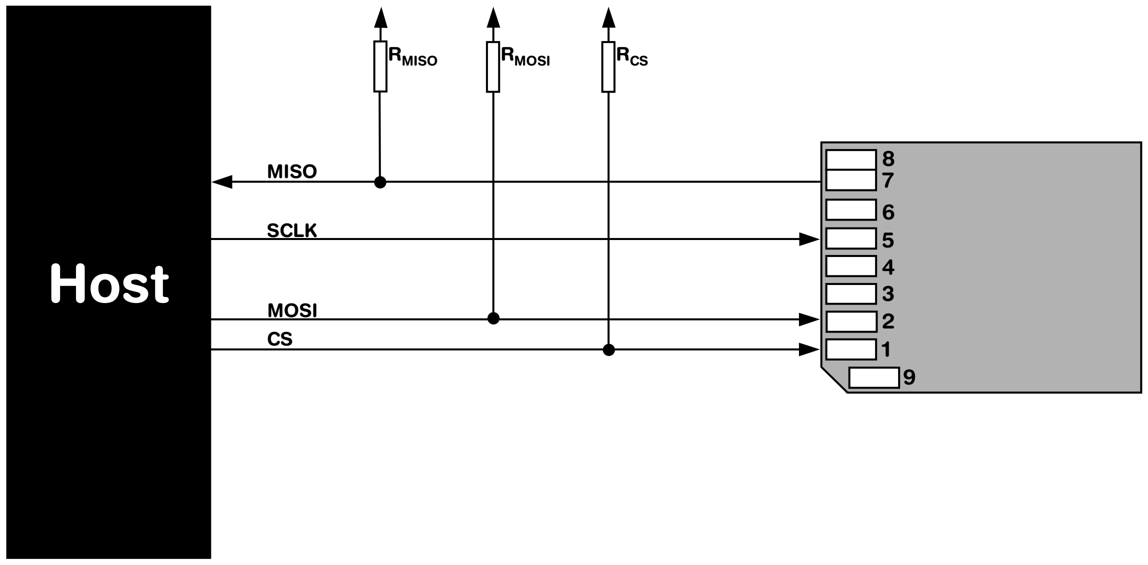
- The data transfer width is 8 bits.
- Data should be output on the falling edge and must remain valid until the next period. Rising edge means data is sampled (i.e. read).
- The bit order requires most significant bit (MSB) to be sent out first.
- Data polarity is normal, which means a logical "1" is represented with a high level on the data line and a logical "0" is represented with low-level.
- MultiMedia & SD cards support different voltage ranges. Initial voltage should be 3.3V.

Power control should be considered when creating designs using the MultiMediaCard and/or SD Card. The ability to have software power control of the cards makes the design more flexible and robust. The host will be able to turn power to the card on or off independent of whether the card is inserted or removed. This can improve card initialization when there is a contact bounce during card insertion. The host waits for a specified time after the card is inserted before powering up the card and starting the initialization process. Also, if the card goes into an unknown state, the host can cycle the power and start the initialization process again. When card access is unnecessary, allowing the host to power-down the bus can reduce the overall power consumption.

Sample schematic for MMC/SD card in Card mode



Sample schematic for MMC/SD card in SPI mode



5.4.2 Theory of operation

The Serial Peripheral Interface (SPI) bus is a very loose de facto standard for controlling almost any digital electronics that accepts a clocked serial stream of bits. SPI operates in full duplex (sending and receiving at the same time).

5.4.3 Fail-safe operation

Unexpected Reset

The data will be preserved.

Power failure

Power failure can be critical: If the card does not have sufficient time to complete a write operation, data may be lost. Countermeasures: make sure the power supply for the card drops slowly.

5.4.4 Wear leveling

MMC / SD cards are controlled by an internal controller. This controller also handles wear leveling. Therefore, the driver does not need to handle wear-leveling.

5.4.5 Configuration

5.4.5.1 Adding the driver to µC/FS

To add the driver use `FS_AddDevice()` with either the driver label `FS_MMC_SPI_Driver` or `FS_MMC_CardMode_Driver`. This function has to be called from within `FS_X_AddDevices()`. Refer to "FS_X_AddDevices()" on page 262 for more information.

Example

SPI mode: `FS_AddDevice(&FS_MMC_SPI_Driver);`

Card mode: `FS_AddDevice(&FS_MMC_CardMode_Driver);`

5.4.5.2 Cyclic redundancy check (CRC)

The cyclic redundancy check (CRC) is a method to produce a checksum. The checksum is a small, fixed number of bits against a block of data. The checksum is used to detect errors after transmission or storage. A CRC is computed and appended before transmission or storage, and verified afterwards by the recipient to confirm that no changes occurred on transit. CRC is a good solution for error detection, but reduces the transmission speed, because a CRC checksum has to be computed for every data block which will be transmitted.

The following functions can be used for controlling CRC calculation in µC/FS.

Function	Description
CRC configuration	
`FS_MMC_ActivateCRC()`	Activates the CRC functionality in SPI mode.
`FS_MMC_DeactivateCRC()`	Deactivates the CRC functionality in SPI mode. By default, CRC is deactivated.

Table 5.40: SPI mode configuration functions

5.4.5.3 FS_MMC_ActivateCRC()

Description

Activates the cyclic redundancy check.

Prototype

```
void FS_MMC_ActivateCRC (void);
```

Additional information

By default, the cyclic redundancy check is deactivated for speed reasons. The driver supports cyclic redundancy check both for all transmissions and just for critical transmissions. You can activate and deactivate the cyclic redundancy check as it fits to the requirements of your application.

5.4.5.4 FS_MMC_DeactivateCRC()

Description

Deactivates the cyclic redundancy check.

Prototype

```
void FS_MMC_DeactivateCRC (void);
```

Additional information

By default, the cyclic redundancy check is deactivated for speed reasons. The driver supports cyclic redundancy check both for all transmissions and just for critical transmissions. You can activate and deactivate the cyclic redundancy check as it fits to the requirements of your application.

5.4.6 Hardware functions - SPI mode

Routine	Explanation
Control line functions	
`FS_MMC_HW_X_EnableCS()`	Activates chip select signal (CS) of the specified card slot.
`FS_MMC_HW_X_DisableCS()`	Deactivates chip select signal (CS) of the specified card slot.
Operation condition detection and adjusting	
`FS_MMC_HW_X_SetMaxSpeed()`	Sets the SPI clock speed. The value is represented in thousand cycles per second (kHz).
`FS_MMC_HW_X_SetVoltage()`	Sets the operating voltage range for the MultiMedia & SD card slot.
Medium status functions	
`FS_MMC_HW_X_IsWriteProtected()`	Checks the status of the mechanical write protection of a SD card.
`FS_MMC_HW_X_IsPresent()`	Checks whether a card is present or not.
Data transfer functions	
`FS_MMC_HW_X_Read()`	Receives a number of bytes from the card.
`FS_MMC_HW_X_Write()`	Sends a number of bytes to the card.

Table 5.41: SPI mode hardware functions

5.4.6.1 FS_MMC_HW_X_EnableCS()

Description

Activates chip select signal (CS) of the specified card slot.

Prototype

```
void FS_MMC_HW_X_EnableCS (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0...N).

Table 5.42: FS_MMC_HW_X_EnableCS() parameter list

Additional Information

The CS signal is used to address a specific card slot connected to the SPI. Enabling is equal to setting the CS line onto low-level.

Example

```
void FS_MMC_HW_X_EnableCS(U8 Unit) {
  SPI_CLR_CS();
}
```

5.4.6.2 FS_MMC_HW_X_DisableCS()

Description

Deactivates chip select signal (CS) of the specified card slot.

Prototype

```
void FS_MMC_HW_X_DisableCS (U8 Unit);
```

Parameter	Meaning
`Unit`	Unit number (0...N).

Table 5.43: FS_MMC_HW_X_DisableCS() parameter list

Additional Information

The CS signal is used to address a specific card slot connected to the SPI. Disabling is equal to setting the CS line to high.

Example

```
void FS_MMC_HW_X_DisableCS(U8 Unit) {
  SPI_SET_CS();
}
```

5.4.6.3 FS_MMC_HW_X_SetMaxSpeed()

Description

Sets the maximum SPI speed. If the hardware is unable to use this speed, a lower freqency can always be selected. The value is given in kHz.

Prototype

```
U16 FS_MMC_HW_X_SetMaxSpeed (U8  Unit,
                             U16 MaxFreq);
```

Parameter	Meaning
Unit	Unit number (0...N).
MaxFreq	Clock speed (kHz) between host and card.

Table 5.44: FS_MMC_HW_X_SetMaxSpeed() parameter list

Return value

Actual frequency in thousand cycles per second (kHz)
0 if the frequency could not be set.

Additional Information

Make sure your SPI interface never generates a higher clock than MaxFreq specifies. You can always run MultiMedia & SD cards at lower or equal, but never on higher frequencies. The initial frequency must be 400kHz or less. If the precise frequency is unknown (typical for implementation using port-pins "bit-banging"), the return value should be less than the maximum frequency, leading to longer timeout values, which is in general unproblematic. You have to return the actual clock speed of your SPI interface, because µC/FS needs the actual frequency to calculate timeout values. If the precise frequency is unknown (typical for implementation using port-pins "bit-banging"), the return value should be less than the maximum frequency, leading to longer timeout values, which is in general unproblematic.

Example using port pins

```
#define MMC_MAXFREQUENCY    400

U16 FS_MMC_HW_X_SetMaxSpeed(U8 Unit, U16 MaxFreq) {
  _Init();
  return MMC_MAXFREQUENCY;    /* We are not faster than this */
}
```

Example using SPI mode

```
U16 FS_MMC_HW_X_SetMaxSpeed(U8 Unit, U16 MaxFreq) {
  U32 InFreq;
  U32 SPIFreq;

  if (MaxFreq < 400) {
    MaxFreq = 400;
  }
  SPIFreq = 1000 * MaxFreq;
  if (SPIFreq >= 200000) {
    InFreq = 48000000;
  }
  _sbcr = (InFreq + SPIFreq - 1) / SPIFreq;
  _InitSPI();
  return MaxFreq;    /* We are not faster than this */
}
```

5.4.6.4 FS_MMC_HW_X_SetVoltage()

Description

Sets the operating voltage range for the MultiMedia & SD card slot.

Prototype

```
char FS_MMC_HW_X_SetVoltage (U8  Unit,
                             U16 Vmin,
                             U16 Vmax);
```

Parameter	Meaning
Unit	Unit number (0...N).
Vmin	Minimum supply voltage in mV.
Vmax	Maximum supply voltage in mV.

Table 5.45: FS_MMC_HW_X_SetVoltage() parameter list

Return value

== 1: Card slot works within the given range.
== 0: Card slot cannot provide a voltage within given range.

Additional Information

The values are in mill volts (mV). 1mV is 0.001V. All cards work with the initial voltage of 3.3V. If you want to save power you can adjust the card slot supply voltage within the given range of Vmin and Vmax.

Example

```
#define FS__MMC_DEFAULTSUPPLYVOLTAGE  3300 /* example means 3.3V */

char FS_MMC_HW_X_SetVoltage(U8 Unit, U16 Vmin, U16 Vmax) {
  /* voltage range check */
  char r;
  if((Vmin <= MMC_DEFAULTSUPPLYVOLTAGE) && (Vmax >= MMC_DEFAULTSUPPLYVOLTAGE)) {
    r = 1;
  } else {
    r = 0;
  }
  return r;
}
```

5.4.6.5 FS_MMC_HW_X_IsWriteProtected()

Description

Checks the status of the mechanical write protection of a SD card.

Prototype

```
char FS_MMC_HW_X_IsWriteProtected (U8 Unit);
```

Parameter	Meaning
`Unit`	Unit number (0...N).

Table 5.46: FS_MMC_HW_X_IsWriteProtected() parameter list

Return value

== 0: If the card is not write protected.
== 1: Means that the card is write protected.

Additional Information

MultiMedia cards do not have mechanical write protection switches and should always return 0. If you are using SD cards, be aware that the mechanical switch does not really protect the card physically from being overwritten; it is the responsibility of the host to respect the status of that switch.

Example

```
char FS_MMC_HW_X_IsWriteProtected(U8 Unit) {
  return 0; /* If the card slot has no write switch detector, return 0 */
}
```

5.4.6.6 FS_MMC_HW_X_IsPresent()

Description

Checks whether a card is present or not.

Prototype

```
char FS_MMC_HW_X_IsPresent (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0...N).

Table 5.47: FS_MMC_HW_X_IsPresent() parameter list

Return value

Return value	Description
FS_MEDIA_STATE_UNKNOWN	The card state is unknown.
FS_MEDIA_NOT_PRESENT	A card is not present.
FS_MEDIA_IS_PRESENT	A card is present.

Table 5.48: FS_MMC_HW_X_IsPresent() - list of return values

Additional Information

Usually, a card slot provides a hardware signal that can be used for card presence determination. The sample code below is for a specific hardware that does not have such a signal. Therefore, the presence of a card is unknown and you have to return `FS_MEDIA_STATE_UNKNOWN`. Then µC/FS tries reading the card to figure out if a valid card is inserted into the slot.

Example

```
char FS_MMC_HW_X_IsPresent(U8 Unit) {
  return FS_MEDIA_STATE_UNKNOWN;
}
```

5.4.6.7 FS_MMC_HW_X_Read()

Description

Receives a number of bytes from the card.

Prototype

```
void FS_MMC_HW_X_Read (U8   Unit,
                       U8 * pData,
                       int  NumBytes);
```

Parameter	Meaning
Unit	Unit number (0...N).
pData	Pointer to a buffer for data to receive.
NumBytes	Number of bytes to receive.

Table 5.49: FS_MMC_HW_X_Read() parameter list

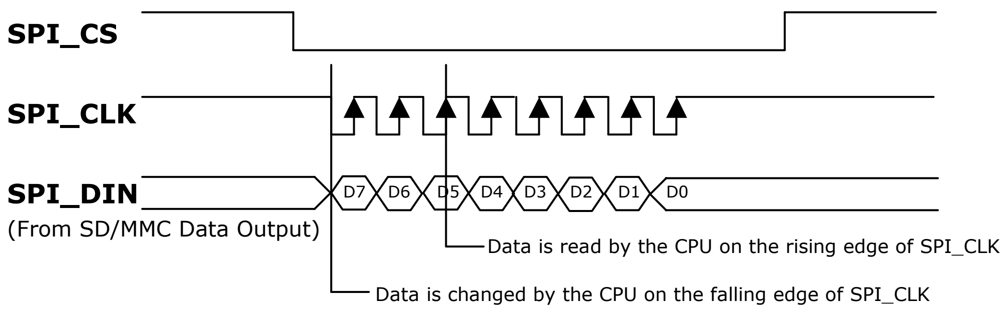
Additional Information

This function is used to read a number of bytes from the card to buffer memory.

Example

```
void FS_MMC_HW_X_Read (U8 Unit, U8 * pData, int NumBytes) {
  do {
    c = 0;
    bpos = 8; /* get 8 bits */
    do {
      SPI_CLR_CLK();
      c <<= 1;
      if (SPI_DATAIN()) {
        c |= 1;
      }
      SPI_SET_CLK();
    } while (--bpos);
    *pData++ = c;
  } while (--NumBytes);
}
```

Timing diagram for read access


SPI_CS
SPI_CLK
SPI_DIN
(From SD/MMC Data Output)
D7
D6
D5
D4
D3
D2
D1
D0
Data is read by the CPU on the rising edge of SPI_CLK
Data is changed by the CPU on the falling edge of SPI_CLK

5.4.6.8 FS_MMC_HW_X_Write()

Description

Sends a number of bytes to the card.

Prototype

```
void FS_MMC_HW_X_Write (U8          Unit,
                        const U8 *  pData,
                        int         NumBytes);
```

Parameter	Meaning
Unit	Unit number (0...N).
pData	Pointer to a buffer that contains the data to be written to the card.
NumBytes	Number of bytes to write.

Table 5.50: FS_MMC_HW_X_Write() parameter list

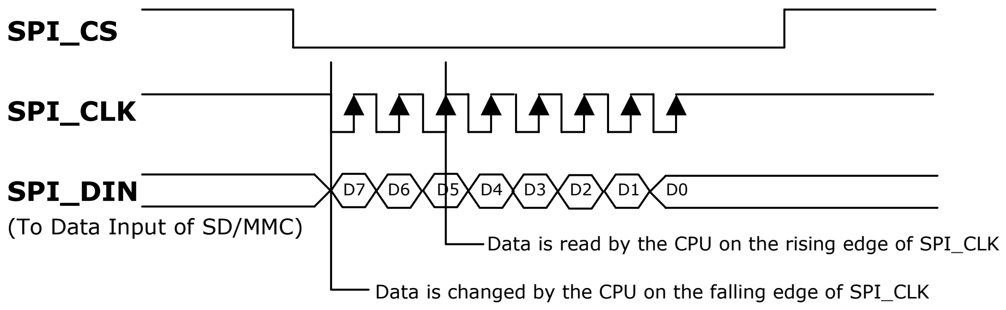
Additional Information

This function is used to send a number of bytes from a memory buffer to the card.

Example

```
void FS_MMC_HW_X_Write(U8 Unit, const U8 * pData, int NumBytes) {
  int i;
  U8 mask;
  U8 data;
  for (i = 0; i < NumBytes; i++) {
    data = pData[i];
    mask = 0x80;
    while (mask) {
      if (data & mask) {
        SPI_SET_DATAOUT();
      } else {
        SPI_CLR_DATAOUT();
      }
      SPI_CLR_CLK();
      SPI_DELAY();
      SPI_SET_CLK();
      SPI_DELAY();
      mask >>= 1;
    }
  }
  SPI_SET_DATAOUT(); /* default state of data line is high */
}
```

Timing diagram for write access


SPI_CS
SPI_CLK
SPI_DIN
(To Data Input of SD/MMC)
D7
D6
D5
D4
D3
D2
D1
D0
Data is read by the CPU on the rising edge of SPI_CLK
Data is changed by the CPU on the falling edge of SPI_CLK

5.4.7 Hardware functions - Card mode

Routine	Explanation
Operation condition detection and adjusting	
`FS_MMC_HW_X_SetMaxSpeed()`	Sets the SD/MMC clock speed. The value is represented in thousand cycles per second (kHz).
`FS_MMC_HW_X_SetResponseTimeOut()`	Sets the card host controller timeout value for receiving response from card.
`FS_MMC_HW_X_SetReadDataTimeOut()`	Sets the card host controller timeout value for receiving data from card.
`FS_MMC_HW_X_SetHWBlockLen()`	Sets the card host controller block size value for a block.
`FS_MMC_HW_X_SetHWNumBlocks()`	Tells the card host controller how many block will be transferred to or received from card.
Medium status functions	
`FS_MMC_HW_X_IsWriteProtected()`	Checks the status of the mechanical write protection of a SD card.
`FS_MMC_HW_X_IsPresent()`	Checks whether a card is present or not.
Data transfer functions	
`FS_MMC_HW_X_GetResponse()`	Retrieves the response after sending a command to the card.
`FS_MMC_HW_X_ReadData()`	Receives a number of bytes from the card.
`FS_MMC_HW_X_SendCmd()`	Sends and setups the controller to send a specific command to card.
`FS_MMC_HW_X_WriteData()`	Writes a number of block to the card.
Time functions	
`FS_MMC_HW_X_Delay()`	Waits for a specific time in ms.

Table 5.51: Card mode hardware functions

5.4.7.1 FS_MMC_HW_X_SetMaxSpeed()

Description

Sets the maximum SD/MMC clock speed. If the hardware is unable to use this speed, a lower freqency can always be selected. The value is given in kHz.

Prototype

```
U16 FS_MMC_HW_X_SetMaxSpeed (U8  Unit,
                             U16 MaxFreq);
```

Parameter	Meaning
Unit	Unit number (0…N).
MaxFreq	Clock speed (kHz) between host and card.

Table 5.52: FS_MMC_HW_X_SetMaxSpeed() parameter list

Return value

Actual frequency in thousand cycles per second (kHz)
0 if the frequency could not be set.

Additional Information

Make sure your SPI interface never generates a higher clock than MaxFreq specifies. You can always run MultiMedia & SD cards at lower or equal, but never on higher frequencies. The initial frequency must be 400kHz or less. If the precise frequency is unknown (typical for implementation using port-pins "bit-banging"), the return value should be less than the maximum frequency, leading to longer timeout values, which is in general unproblematic. You have to return the actual clock speed of your SPI interface, because µC/FS needs the actual frequency to calculate timeout values. If the precise frequency is unknown (typical for implementation using port-pins "bit-banging"), the return value should be less than the maximum frequency, leading to longer timeout values, which is in general unproblematic.

Example

```
U16 FS_MMC_HW_X_SetMaxSpeed(U8 Unit, U16 MaxFreq) {
  U32 Prediv;
  U32 Rate;

  if (Freq <= 400) {
    Prediv =  8;     //  HCLK / 8, where HCLK is 100MHz. -> SDClock = 12.5 MHz
    Rate   =  5;     //  Card clock frequency = SDClock / (1 << Rate) = 390kHz.
  } else {
    Prediv =  5;     // HCLK / 5, where HCLK is 100MHz, SDClock = 20 MHz
    Rate   =  0;     //  Card clock frequency = SDClock / (1 << Rate) = 20 MHz.
  }
  __SDMMC_PREDIV    = (1 << 5)         // Use Poll mode instead of DMA
                    | (1 << 4)         // Enable the Controller
                    | (Prediv & 0x0f); // Set the predivisor value
  __SDMMC_RATE      = Rate;             // Set rate value
  return Freq;
}
```

5.4.7.2 FS_MMC_HW_X_SetResponseTimeOut()

Description

Sets the card host controller timeout value for receiving response from card.

Prototype

```
void   FS_MMC_HW_X_SetResponseTimeOut (U8  Unit,
                                        int Value);
```

Parameter	Meaning
Unit	Unit number (0...N).
Value	Number of MMC clock cycles to wait before a response timeout occurs.

Table 5.53: FS_MMC_HW_X_Set_ResponseTimeOut() parameter list

Additional Information

This function sets the card host controller timeout value in MMC clock cycles before the card host controller sets a timeout condition when receiving a response from card.

Example

```
void FS_MMC_HW_X_SetResponseTimeOut(U8 Unit, int Value) {
  __SDMMC_RES_TO    = Value;           // Set the timeout for Card Response
}
```

5.4.7.3 FS_MMC_HW_X_SetReadDataTimeOut()

Description

Sets the card host controller timeout value for receiving data from card.

Prototype

```
void FS_MMC_HW_X_SetReadDataTimeOut (U8  Unit,
                                     int Value);
```

Parameter	Meaning
Unit	Unit number (0...N).
Value	Number of MMC clock cycles to wait before a Read Data timeout occurs.

Table 5.54: FS_MMC_HW_X_SetReadDataTimeOut() parameter list

Additional Information

This function sets the card host controller timeout value in MMC clock cycles before the card host controller sets a timeout condition when receiving data from card.

Example

```
void FS_MMC_HW_X_SetReadDataTimeOut(U8 Unit, int Value) {
  __SDMMC_READ_TO    = Value;           // Set the read timeout
}
```

5.4.7.4 FS_MMC_HW_X_SetHWBlockLen()

Description

Sets the card host controller block size value for a block.

Prototype

```
void   FS_MMC_HW_X_SetHWBlockLen (U8  Unit,
                                  U16 BlockSize);
```

Parameter	Meaning
Unit	Unit number (0...N).
BlockSize	Block size given in number of bytes.

Table 5.55: FS_MMC_HW_X_SetHWBlockLen() parameter list

Additional Information

Card host controller sends data to or receives data from the card in block chunks. This function typically sets the card host controller's block length register.

Example

```
void FS_MMC_HW_X_SetHWBlockLen(U8 Unit, U16 BlockSize) {
  __SDMMC_BLK_LEN  = BlockSize;
}
```

5.4.7.5 FS_MMC_HW_X_SetHWNumBlocks()

Description

Tells the card host controller how many block will be transferred to or received from card.

Prototype

```
void FS_MMC_HW_X_SetHWNumBlocks (U8  Unit,
                                 U16 NumBlocks);
```

Parameter	Meaning
Unit	Unit number (0...N).
NumBlocks	Number of blocks to be transferred.

Table 5.56: FS_MMC_HW_X_SetHWNumBlocks() parameter list

Additional Information

Before sending the command to read or write data from or to the card. This functions tells the card host controller, how many blocks need to be transferred/received.

Example

```
void FS_MMC_HW_X_SetHWNumBlocks(U8 Unit, U16 NumBlocks) {
  __SDMMC_NUM_BLK   = NumBlocks;
}
```

5.4.7.6 FS_MMC_HW_X_IsWriteProtected()

Description

Checks the status of the mechanical write protection of a SD card.

Prototype

```
char FS_MMC_HW_X_IsWriteProtected (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0...N).

Table 5.57: FS_MMC_HW_X_IsWriteProtected() parameter list

Return value

== 0: If the card is not write protected.
== 1: Means that the card is write protected.

Additional Information

MultiMedia cards do not have mechanical write protection switches and should always return 0. If you are using SD cards, be aware that the mechanical switch does not really protect the card physically from being overwritten; it is the responsibility of the host to respect the status of that switch.

Example

```
char FS_MMC_HW_X_IsWriteProtected(U8 Unit) {
  return 0; /* Card slot has no write switch detector, return 0 */
}
```

5.4.7.7 FS_MMC_HW_X_IsPresent()

Description

Checks whether a card is present or not.

Prototype

```
char FS_MMC_HW_X_IsPresent (U8 Unit);
```

Parameter	Meaning
`Unit`	Unit number (0...N).

Table 5.58: FS_MMC_HW_X_IsPresent() parameter list

Return value

Return value	Meaning
`FS_MEDIA_STATE_UNKNOWN`	State of the media is unknown.
`FS_MEDIA_NOT_PRESENT`	No card is present.
`FS_MEDIA_IS_PRESENT`	Card is present.

Table 5.59: FS_MMC_HW_X_IsPresent() - list of return values

Additional Information

Usually, a card slot provides a hardware signal that can be used for card presence determination. The sample code below is for a specific hardware that does not have such a signal. Therefore, the presence of a card is unknown and you have to return `FS_MEDIA_STATE_UNKNOWN`. Then µC/FS tries reading the card to figure out if a valid card is inserted into the slot.

Example

```
char FS_MMC_HW_X_IsPresent(U8 Unit) {
  __GPIO_PFDD &=  ~(1 << 5);  // Set PE.5 as input for card detect signal
  return ((__GPIO_PFD >> 5) & 1) ? FS_MEDIA_NOT_PRESENT : FS_MEDIA_IS_PRESENT;
}
```

5.4.7.8 FS_MMC_HW_X_GetResponse()

Description

Retrieves the response after sending a command to the card.

Prototype

```
int FS_MMC_HW_X_GetResponse (U8       Unit,
                             void   * pBuffer,
                             U32      Size);
```

Parameter	Meaning
Unit	Unit number (0...N).
pBuffer	Pointer to a buffer for data to receive the response.
NumBytes	Number of bytes to receive.

Table 5.60: FS_MMC_HW_X_GetResponse() parameter list

Return value

Return value	Meaning
FS_MMC_CARD_NO_ERROR	All data have been read successfully.
FS_MMC_CARD_RESPONSE_TIMEOUT	Card did not send the response in appropriate time.
FS_MMC_CARD_READ_CRC_ERROR	The received response failed the CRC check of card host controller.

Table 5.61: FS_MMC_HW_X_GetResponse() - list of return values

Additional Information

This function is used to read the card's response to buffer memory.

Example

```
int FS_MMC_HW_X_GetResponse(U8 Unit, void *pBuffer, U32 Size) {
  U16 * pResponse;
  U32   Index;
  U32   Status;

  pResponse = (U16 *) pBuffer;
  // Wait for response
  while (1) {
    Status = __SDMMC_STATUS;
    if (Status & MMC_STATUS_CLOCK_DISABLED) {
      _StartMMCClock(Unit);
    }
    if (Status & MMC_STATUS_END_COMMAND_RESPONSE) {
      break;
    }
    if (Status & MMC_STATUS_RESPONSE_TIMEOUT) {
      return FS_MMC_CARD_RESPONSE_TIMEOUT;
    }
    if (Status & MMC_STATUS_RESPONSE_CRC_ERROR) {
      return FS_MMC_CARD_RESPONSE_CRC_ERROR;
    }
  }
  // Read the necessary number of response words from the response FIFO
  for (Index = 0; Index < (Size/ 2); Index++) {
    pResponse[Index] = __SDMMC_RES_FIFO;
  }
  return FS_MMC_CARD_NO_ERROR;
}
```

5.4.7.9 FS_MMC_HW_X_ReadData()

Description

Receives a number of bytes from the card.

Prototype

```
int    FS_MMC_HW_X_ReadData (U8       Unit,
                             void *   pBuffer,
                             unsigned NumBytes,
                             unsigned NumBlocks);
```

Parameter	Meaning
Unit	Unit number (0...N).
pBuffer	Pointer to a buffer for data to receive the response.
NumBytes	Number of bytes to receive.
NumBlocks	Number of blocks to receive.

Table 5.62: FS_MMC_HW_X_ReadData() parameter list

Return value

Return value	Meaning
FS_MMC_CARD_NO_ERROR	All data have been read successfully.
FS_MMC_CARD_READ_TIMEOUT	Card did not send the data in appropriate time.
FS_MMC_CARD_READ_CRC_ERROR	The received response failed the CRC check of card host controller.

Table 5.63: FS_MMC_HW_X_ReadData() - list of return values

Additional Information

This function is used to read the data is coming from MMC/SD card to the host controller through the DAT line or DAT[0:3] lines.

Example

```
int  FS_MMC_HW_X_ReadData  (U8  Unit,  void  *  pBuffer,  unsigned  NumBytes,  unsigned
NumBlocks) {
  U16 * pBuf = (U16 *)pBuffer;
  int i;
  do {
    i = 0;
    // Wait until transfer is complete
    while ((__SDMMC_STATUS & MMC_STATUS_FIFO_FULL) == 0);
    if (__SDMMC_STATUS & MMC_STATUS_READ_CRC_ERROR) {
      return FS_MMC_CARD_READ_CRC_ERROR;
    }
    if (__SDMMC_STATUS & MMC_STATUS_READDATA_TIMEOUT) {
      return FS_MMC_CARD_READ_TIMEOUT;
    }
    // Continue reading data until FIFO is empty
    while(((__SDMMC_STATUS & MMC_STATUS_FIFO_EMPTY) == 0) && (i < (NumBytes >> 1))) {
      // Any data in the FIFO
      if ((__SDMMC_STATUS & MMC_STATUS_FIFO_EMPTY) == 0) {
        *pBuf = __SDMMC_DATA_FIFO;
        pBuf++;
        i++;
      }
    }
  } while (--NumBlocks);
  return 0;
}
```

5.4.7.10 FS_MMC_HW_X_SendCmd()

Description

Sends and setups the controller to send a specific command to card.

Prototype

```
void    FS_MMC_HW_X_SendCmd (U8         Unit,
                             unsigned Cmd,
                             unsigned CmdFlags,
                             unsigned ResponseType,
                             U32        Arg);
```

Parameter	Meaning
Unit	Unit number (0...N).
Cmd	Command to be sent to the card.
CmdFlags	Additional command flags, that are necessary for this command.
ResponseType	Specifies the response format that is expected after sending this command.
Arg	Argument sent with command.

Table 5.64: FS_MMC_HW_X_SendCmd() parameter list

Additional Information

This function should send the command specified by Cmd. Each command may have some command flags: One or a combination of these is possible:

Command Flag	Meaning
FS_MMC_CMD_FLAG_DATATRANSFER	This flags tells the card controller, that the sent command initiate a data transfer.
FS_MMC_CMD_FLAG_WRITETRANSFER	This flags tells the card controller, that the sent command initiate a data transfer and will write to the card.
FS_MMC_CMD_FLAG_SETBUSY	The card may be in busy state after sending this command. The card host controller may wait after the card ready for next command.
FS_MMC_CMD_FLAG_INITIALIZE	The card host controller should send the initialization sequence to the card.
FS_MMC_CMD_FLAG_USE_SD4MODE	This tells the card host controller to use all 4 DAT line rather than DAT line. Note, that this command flag is only set when FS_MMC_SUPPORT_4BIT_MODE is set.
FS_MMC_CMD_FLAG_STOP_TRANS	The card host controller shall stop transferring data to the card.
FS_MMC_CMD_FLAG_DATATRANSFER	This flags tells the card controller, that the sent command initiate a data transfer.
FS_MMC_CMD_FLAG_WRITETRANSFER	This flags tells the card controller, that the sent command initiate a data transfer and will write to the card.
FS_MMC_CMD_FLAG_SETBUSY	The card may be in busy state after sending this command. The card host controller may wait after the card ready for next command.

Table 5.65: FS_MMC_HW_X_SendCmd() - list of possible commands

After sending a command to the MMC/SD card, the card may respond to the command. The type/format of the expected response can be one of the following:

Response Type	Meaning
FS_MMC_RESPONSE_FORMAT_NONE	No response is expected from card.
FS_MMC_RESPONSE_FORMAT_R1	Response type 1 is expected from card. (48 Bit data stream is sent by card through the CMD line.)
FS_MMC_RESPONSE_FORMAT_R2	Response type 2 is expected from card. (136 Bit data stream is sent by card through the CMD line.)
FS_MMC_RESPONSE_FORMAT_R3	Response type 3 is expected from card. (48 Bit data stream is sent by card through the CMD line.)

Table 5.66: FS_MMC_HW_X_SendCmd() - list of possible responses

If the specified command expects a response, `FS_MMC_HW_X_GetResponse()` will be called after `FS_MMC_HW_X_SendCmd()`.

Example

```
void FS_MMC_HW_X_SendCmd(U8 Unit, unsigned Cmd, unsigned CmdFlags,
                         unsigned ResponseType, U32 Arg) {
  U32 CmdCon;
  _StopMMCClock(Unit);
  CmdCon = ResponseType;
  if (CmdFlags & FS_MMC_CMD_FLAG_DATATRANSFER) { /* If data transfer */
    CmdCon |= (1 << 8)     /* Set big endian flag for data transfers
                              since this is how the data is in the 16-bit fifo */
           |  (1 << 2);    // Set DATA_EN
  }
  if (CmdFlags & FS_MMC_CMD_FLAG_WRITETRANSFER) {   /* Abort transfer ? */
    CmdCon |= (1 << 3);    // Set WRITE bit
  }
  if (CmdFlags & FS_MMC_CMD_FLAG_SETBUSY) {        /* Set busy ? */
    CmdCon |= (1 << 5);    // Set ABORT bit
  }
  if (CmdFlags & FS_MMC_CMD_FLAG_INITIALIZE) {   /* Init ? */
    CmdCon |= (1 << 6);    // Set ABORT bit
  }
  if (CmdFlags & FS_MMC_CMD_FLAG_USE_SD4MODE) {   /* 4 bit mode ? */
    CmdCon |= (1 << 7);    // Set WIDE bit
  }
  if (CmdFlags & FS_MMC_CMD_FLAG_STOP_TRANS) {   /* Abort transfer ? */
    CmdCon |= (1 << 13);    // Set ABORT bit
  }
  __SDMMC_CMD       = Cmd;
  __SDMMC_CMDCON    = CmdCon;
  __SDMMC_ARGUMENT = Arg;
  _StartMMCClock(Unit);
}
```

5.4.7.11 FS_MMC_HW_X_WriteData()

Description

Writes a number of block to the card.

Prototype

```
int FS_MMC_HW_X_WriteData (U8             Unit,
                           const void * pBuffer,
                           unsigned     NumBytes,
                           unsigned     NumBlocks);
```

Parameter	Meaning
Unit	Unit number (0...N).
pBuffer	Pointer to a buffer for data to send.
NumBytes	Number of bytes for each block to send.
NumBlocks	Number of blocks to send.

Table 5.67: FS_MMC_HW_X_WriteData() parameter list

Return value

Return Flag	Meaning
FS_MMC_CARD_NO_ERROR	All data have been sent successfully and card has programmed the data.
FS_MMC_CARD_WRITE_CRC_ERROR	During the data transfer to the card a CRC error occurred.

Table 5.68: FS_MMC_HW_X_WriteData() - list of return values

Additional Information

This function is used to write a specified number of blocks to the card. Each block is NumBytes long.

Example

```
int FS_MMC_HW_X_WriteData(U8 Unit, const void * pBuffer,
                           unsigned NumBytes, unsigned NumBlocks) {
  int         i;
  const U16 * pBuf;
  pBuf = (const U16 *)pBuffer;
  do {
    while((__SDMMC_STATUS & MMC_STATUS_FIFO_EMPTY) == 0);
    for (i = 0; i < (NumBytes >> 1); i++) {
      __SDMMC_DATA_FIFO =  *pBuf++;
    }
    _StartMMCClock(Unit);
    if (__SDMMC_STATUS & MMC_STATUS_WRITE_CRC_ERROR) {
      return FS_MMC_CARD_WRITE_CRC_ERROR;
    }
  } while (--NumBlocks);
  // Wait until transfer operation has ended
   while ((__SDMMC_STATUS & MMC_STATUS_DATA_TRANFER_DONE) == 0);
  // Wait until write operation has ended
  while ((__SDMMC_STATUS & MMC_STATUS_DATA_PROGRAM_DONE) == 0);
  return 0;
}
```

5.4.7.12 FS_MMC_HW_X_Delay()

Description

Waits for a specific time in ms.

Prototype

```
void FS_MMC_HW_X_Delay (int ms);
```

Parameter	Meaning
ms	Milliseconds to wait.

Table 5.69: FS_MMC_HW_X_Delay() parameter list

Additional Information

The delay specified is a minimum delay. The actual delay is permitted to be longer. This can be helpful when using an RTOS. Every RTOS has a delay API function, but the accuracy is typically 1 tick, which is 1 ms in most cases. Therefore, a delay of 1 tick is typically between 0 and 1 ms. To compensate for this, the equivalent of 1 tick (typically 1) should be added to the delay parameter before passsing it to an RTOS delay function.

Example

```
void   FS_MMC_HW_X_Delay(int ms) {
  OS_Delay(ms + 1);   // Make sure we delay at least <ms> milliseconds
}
```

5.4.8 Additional information

For more technical details about MultiMedia & SD cards, check the documents and specifications available on the following internet web pages:

http://www.mmca.org/

http://www.sdcard.org/

5.4.9 Resource usage

5.4.9.1 ROM usage

The ROM usage depends on the compiler options, the compiler version and the used CPU. The memory requirements of the MMC/SD driver have been measured on a system as follows: ARM7, IAR Embedded workbench V4.41A, Thumb mode, Size optimization.

Module	ROM [Kbytes]
µC/FS SD card SPI mode driver	2.8
µC/FS SD card card mode driver	2.6

5.4.9.2 Static RAM usage

Static RAM usage is the amount of RAM required by the driver for variables inside of the driver. The number of bytes can be seen in a list file.

Static RAM usage of the SD card driver in SPI mode: 12 bytes
Static RAM usage of the SD card driver in card mode: 24 bytes

5.4.10 FAQs

None.

5.4.11 Troubleshooting

If the driver test fails or if the card cannot be accessed at all, please follow the trouble shooting guidelines below.

5.4.11.1 SPI mode troubleshooting guide

Verify SPI configuration

If an SPI is used, you should verify that it is set up as follows:

- 8 bits per transfer
- Most significant bit first
- Data changes on falling edge
- Data is ledged on rising edge.

Verify signals during initialization of the card

The oscilloscope has been set up as follows:

Color	Description
RED	MOSI - Master Out Slave In (Pin 2)
PURPLE	MISO - Master In Slave Out (Pin 7)
GREEN	CLK - Clock (Pin 5)
YELLOW	CS - Chip Select (Pin 1)

Table 5.70: Screenshot descriptions

Trigger: Single, falling edge of CS

To check if your implementation of the hardware layer works correct, compare your output of the relevant lines (SCLK, CS, MISO, MOSI) with the correct output which is shown in the following screenshots. The output of your card should be similar.

In the example, MISO has a pull-up and a pull-down of equal value. This means that the MISO signal level is at 50% (1.65V) when the output of the card is inactive. On other target hardware, the inactive level can be low (in case a pull-down is used) or high (if a pull-up is used).

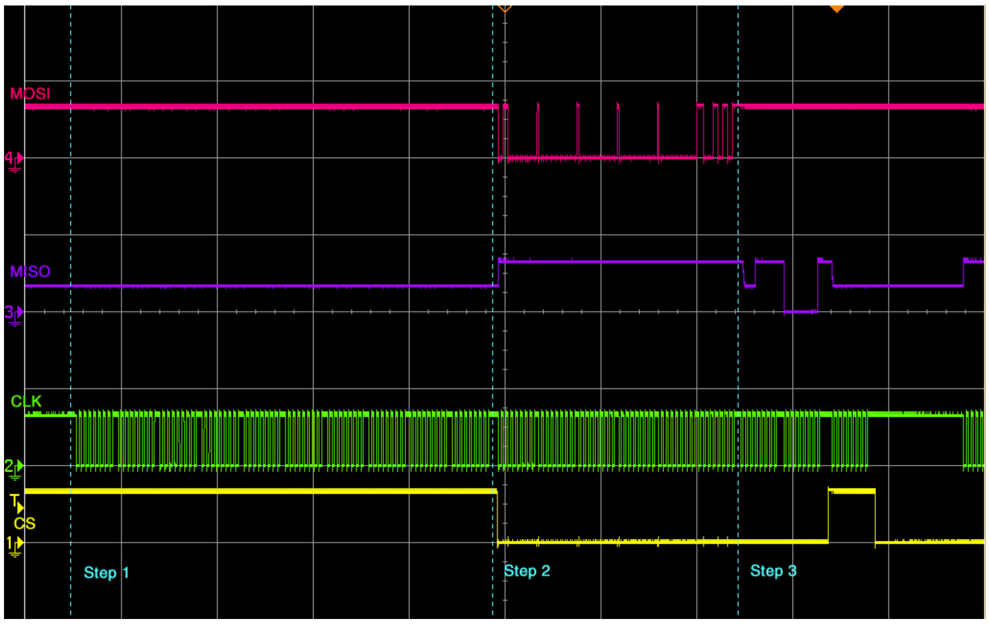
Initial communication sequence

The initial communication sequence consists of the following three parts:

1. Outputs 10 dummy bytes with CS disabled, MOSI = 1.
2. Sets CS low and send a 6-byte command (`GO_IDLE_STATE` command).
3. Receives two byte, set CS high and output 1 dummy byte with CS disabled, MOSI = 1.

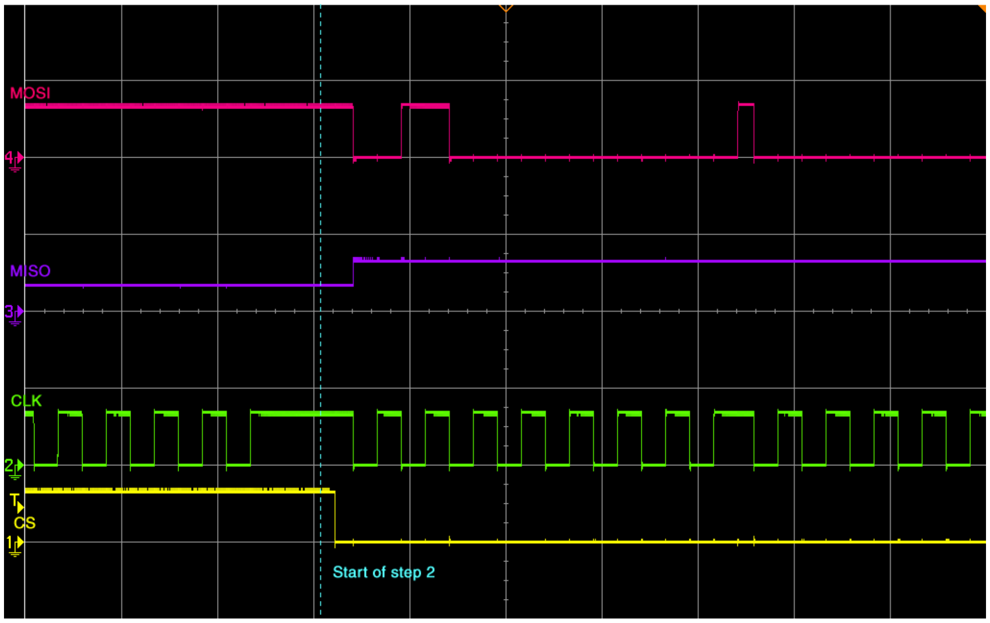
Overview

The screenshot shows the data flow of a correct initialization. It has been captured with an oscilloscope.



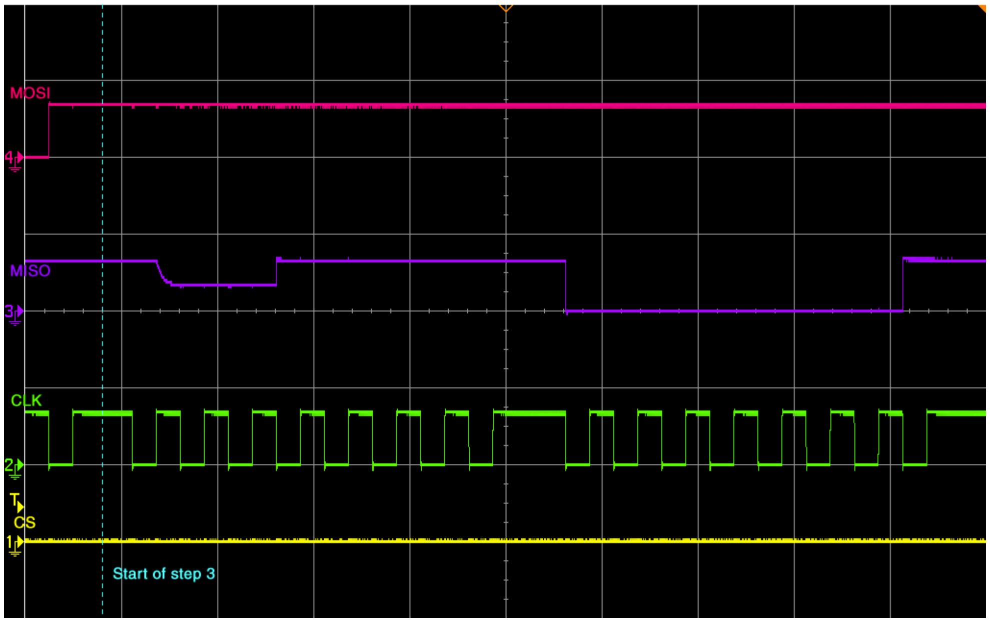
Verify command transfer (Step 2)

After sending 8 dummy bytes to the card, CS is selected and the GO_IDLE_STATE command is sent to the card. The GO_IDLE_STATE command is the reset commandand. It sets the card into idle state regardless of the current card state.



Check output of card (Step 3)

The card responses to the command with two bytes. The SD Card Association defines that the first byte of the response should always be ignored. The second byte is the answer from the card. The answer to GO_IDLE_STATE command should be 0x01. This means that the card is in idle state.

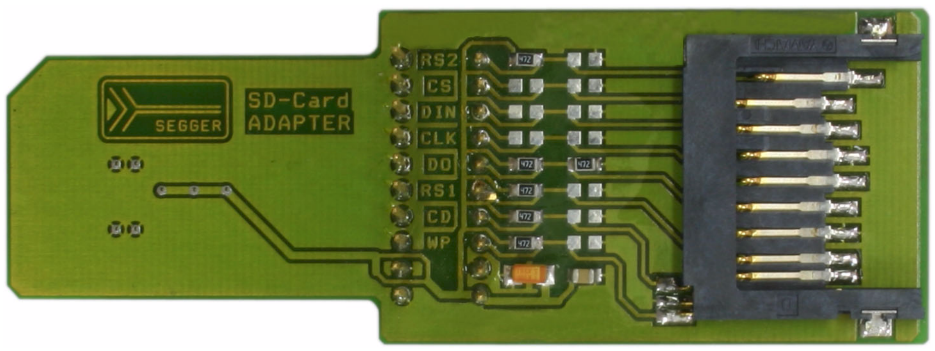


If your card does not return 0x01, check your initialization sequence. After the command sequence CS has to be deselected.

Adapter

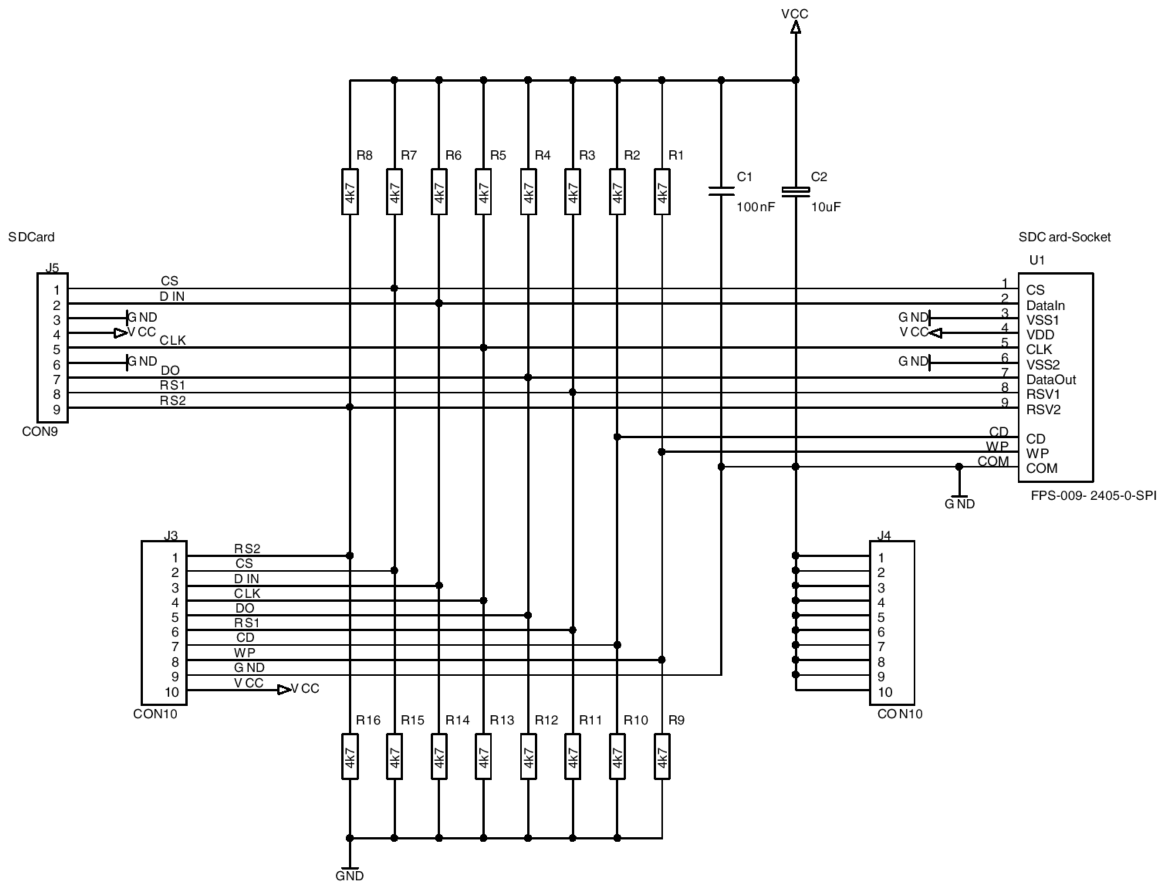
On some evaluation boards the pins required for measuring are not accessible, so that an oscilloscope or logic analyzer cannot capture the outputs. An adapter which can be inserted between the card slot and the card, is the best solution in those situations.

An example adapter is shown below and is available from Micrium.



Adapter schematics

Use the schematic below to build an compatible adapter.



5.5 CompactFlash card & IDE driver

µC/FS supports the use of CompactFlash & IDE devices. An optional generic drivers for CompactFlash & IDE devices is available.

To use the driver with your specific hardware, you will have to provide basic I/O functions for accessing the ATA I/O registers. This section describes all these routines.

5.5.1 Supported Hardware

µC/FS's CompactFlash & IDE device driver can be used to access most ATA HD drives or CompactFlash storage cards also known as CF using true IDE or Memory card mode.

True IDE mode pin functions

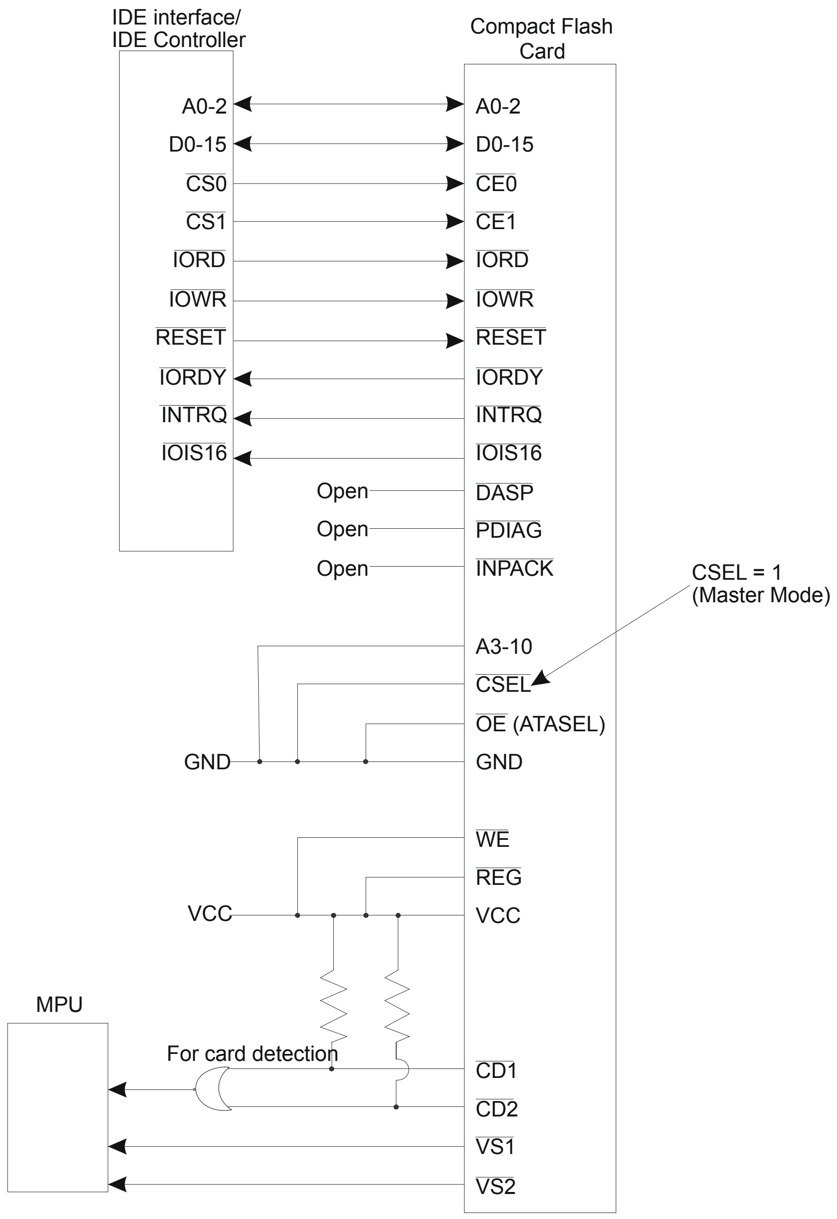
Signal name	Dir	Pin	Description
A2-A0	I	18, 19, 20	Only A[2:0] are used to select one of eight registers in the Task File, the remaining address lines should be grounded by the host.
PDIAG	I/O	46	This input / output is the Pass Diagnostic signal in the Master / Slave handshake protocol.
DASP	I/O	45	This input/output is the Disk Active/Slave Present signal in the Master/Slave handshake protocol.
$\overline{CD1}$, $\overline{CD2}$	O	26, 25	These Card Detect pins are connected to ground on the CompactFlash Storage Card or CF+ Card. They are used by the host to determine that the CompactFlash Storage Card or CF+ Card is fully inserted into its socket.
$\overline{CS0}$, $\overline{CS1}$	I	7, 32	CS0 is the chip select for the task file registers while CS1 is used to select the Alternate Status Register and the Device Control Register.
CSEL	I	39	This internally pulled up signal is used to configure this device as a Master or a Slave when configured in True IDE Mode. When this pin is grounded, the device is configured as a Master. When the pin is open, the device is configured as a Slave.
D15 - D00	I/O	27 - 31 47 - 49 2 - 6 21 - 23	All Task File operations occur in byte mode on the low order bus D00-D07 while all data transfers are 16 bit using D00-D15.
GND	--	1, 5	Ground.
IORD	I	34	This is an I/O Read strobe generated by the host. This signal gates I/O data onto the bus from the CompactFlash Storage Card or CF+ Card when the card is configured to use the I/O interface.
IOWR	I	35	I/O Write strobe pulse is used to clock I/O data on the Card Data bus into the CompactFlash Storage Card or CF+ Card controller registers when the CompactFlash Storage Card or CF+ Card is configured to use the I/O interface. The clocking will occur on negative to positive edge of the signal (trailing edge).
$\overline{OE}$ (ATA SEL)	I	9	To enable True IDE Mode this input should be grounded by the host.
INTRQ	O	37	Signal is the active high interrupt request to the host.
REG	I	44	This input signal is not used and should be connected to VCC by the host.
RESET	I	41	This input pin is the active low hardware reset from the host.
VCC	--	13, 38	+5V, +3.3V power.
$\overline{VS1}$, $\overline{VS2}$	O	33, 4	Voltage Sense Signals. -VS1 is grounded so that the CompactFlash Storage Card or CF+ Card CIS can be read at 3.3 volts and -VS2 is reserved by PCMCIA for a secondary voltage.

Table 5.71: True IDE pin functions

Signal name	Dir	Pin	Description
IORDY	O	42	This output signal may be used as IORDY.
WE	I	36	This input signal is not used and should be connected to VCC by the host.
IOIS16	O	24	This output signal is asserted low when the device is expecting a word data transfer cycle.

Table 5.71: True IDE pin functions(Continued)

Sample block schematic



Memory card mode pin functions

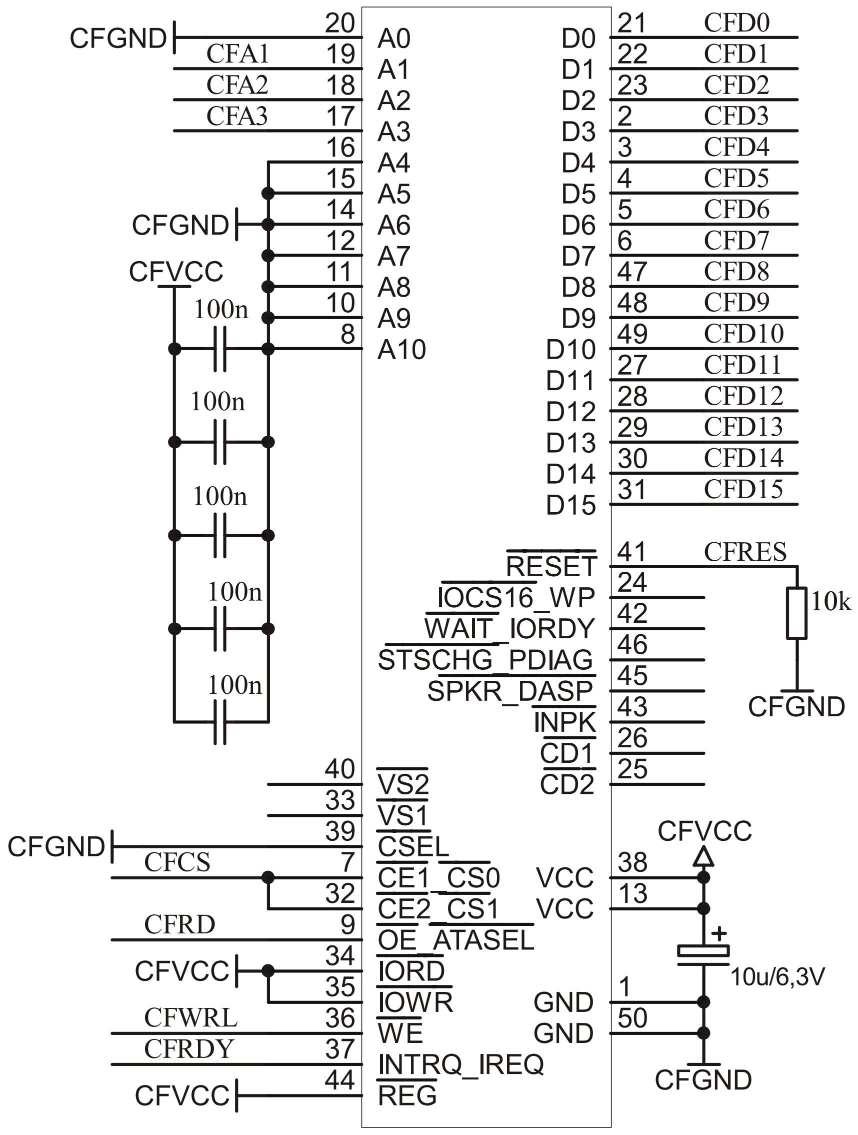
Signal name	Dir	Pin	Description
A10 - A0	I	8, 10, 11, 12, 14, 15, 16, 17, 18, 19, 20	These address lines along with the -REG signal are used to select the following: the I/O port address registers within the CompactFlash Storage Card or CF+ Card, the memory mapped port address registers within the CompactFlash Storage Card or CF+ Card, a byte in the card's information structure and its configuration control and status registers.
BVD1	I/O	46	This signal is asserted high, as BVD1 is not supported.
BVD2	I/O	45	This signal is asserted high, as BVD2 is not supported.
$\overline{\text{CD1}}$, $\overline{\text{CD2}}$	O	26, 25	These Card Detect pins are connected to ground on the CompactFlash Storage Card or CF+ Card. They are used by the host to determine that the CompactFlash Storage Card or CF+ Card is fully inserted into its socket.
$\overline{\text{CE1}}$, $\overline{\text{CE2}}$	I	7, 32	These input signals are used both to select the card and to indicate to the card whether a byte or a word operation is being performed. -CE2 always accesses the odd byte of the word. We recommend connecting these pins together.
CSEL	I	39	This signal is not used for this mode, but should be grounded by the host.
D15 - D00	I/O	27 - 31 47 - 49 2 - 6 21 - 23	These lines carry the Data, Commands and Status information between the host and the controller. D00 is the LSB of the Even Byte of the Word. D08 is the LSB of the Odd Byte of the Word.
GND	--	1, 5	Ground.
INPACK	O	43	This signal is not used in this mode.
IORD	I	34	This signal is not used in this mode.
IOWR	I	35	This signal is not used in this mode.
$\overline{\text{OE}}$ (ATA SEL)	I	9	This is an Output Enable strobe generated by the host interface. It is used to read data from the CompactFlash Storage Card or CF+ Card in Memory Mode and to read the CIS and configuration registers.

Table 5.72: Pin functions in memory card mode

Signal name	Dir	Pin	Description
READY	O	37	In Memory Mode, this signal is set high when the CompactFlash Storage Card or CF+ Card is ready to accept a new data transfer operation and is held low when the card is busy. At power up and at Reset, the READY signal is held low (busy) until the CompactFlash Storage Card or CF+ Card has completed its power up or reset function. No access of any type should be made to the CompactFlash Storage Card or CF+ Card during this time.Note, however, that when a card is powered up and used with +RESET continuously disconnected or asserted, the reset function of this pin is disabled and consequently the continuous assertion of +RESET will not cause the READY signal to remain continuously in the busy state.
REG	I	44	This signal is used during Memory Cycles to distinguish between Common Memory and Register (Attribute) Memory accesses. High for Common Memory, Low for Attribute Memory. To use it with µC/FS, this signal should be high.
RESET	I	41	When the pin is high, this signal Resets the CompactFlash Storage Card or CF+ Card. The CompactFlash Storage Card or CF+ Card is reset only at power up if this pin is left high or open from power-up.
VCC	--	13, 38	+5 V, +3.3 V power.
$\overline{\text{VS1}}$, $\overline{\text{VS2}}$	O	33, 4	Voltage Sense Signals. -VS1 is grounded so that the CompactFlash Storage Card or CF+ Card CIS can be read at 3.3 volts and -VS2 is reserved by PCMCIA for a secondary voltage.
WAIT	O	42	The -WAIT signal is driven low by the CompactFlash Storage Card or CF+ Card to signal the host to delay completion of a memory or I/O cycle that is in progress.
WE	I	36	This is a signal driven by the host and used for strobing memory write data to the registers of the CompactFlash Storage Card or CF+ Card when the card is configured in the memory interface mode.
WP	O	24	The CompactFlash Storage Card or CF+ Card does not have a write protect switch. This signal is held low after the completion of the reset initialization sequence.

Table 5.72: Pin functions in memory card mode(Continued)

Sample block schematic



5.5.2 Theory of operation

5.5.2.1 CompactFlash

CompactFlash is a mechanically small, removable mass storage device. The CompactFlash Storage Card contains a single chip controller and flash memory module(s) in a matchbox-sized package with a 50-pin connector consisting of two rows of 25 female contacts each on 50 mil (1.27 mm) centers. The controller interfaces with a host system allowing data to be written to and read from the flash memory module(s).

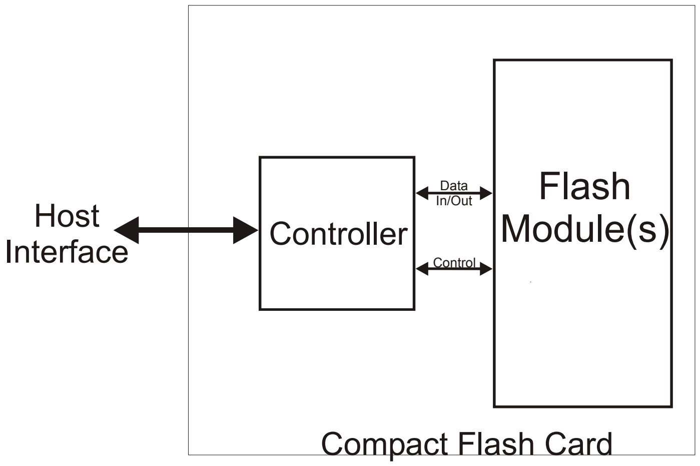


Figure 5.1: CompactFlash schematic

There are two different Compact Flash Types, namely CF Type I and CF Type II.
The only difference between CF Type I and CF Type II cards is the card thickness. CF Type I is 3.3 mm thick and CF Type II cards are 5mm thick. A CF Type I card will operate in a CF Type I or CF Type II slot. A CF Type II card will only fit in a CF Type II slot. The electrical interfaces are identical. CompactFlash is available in both CF Type I and CF Type II cards, though predominantly in CF Type I cards. The Microdrive is a CF Type II card. Most CF I/O cards are CF Type I, but there are some CF Type II I/O cards.

CompactFlash cards are designed with flash technology, a nonvolatile storage solution that does not require a battery to retain data indefinitely.
The CompactFlash card specification version 2.0 supports data rates up to 16MB/sec and capacities up to 137GB.
CF cards consume only five percent of the power required by small disk drives.

CompactFlash cards support both 3.3V and 5V operation and can be interchanged between 3.3V and 5V systems. This means that any CF card can operate at either voltage. Other small form factor flash cards may be available to operate at 3.3V or 5V, but any single card can operate at only one of the voltages.
CF+ data storage cards are also available using magnetic disk (IBM Microdrive).

Modes of operation (interface modes)

Compact Flash cards can operate in three modes:

- Memory card mode
- I/O Card mode
- True IDE mode

Supported modes of operation (interface modes)

Currently, TRUE IDE and MEMORY CARD mode are supported.

Memory card mode pin functions

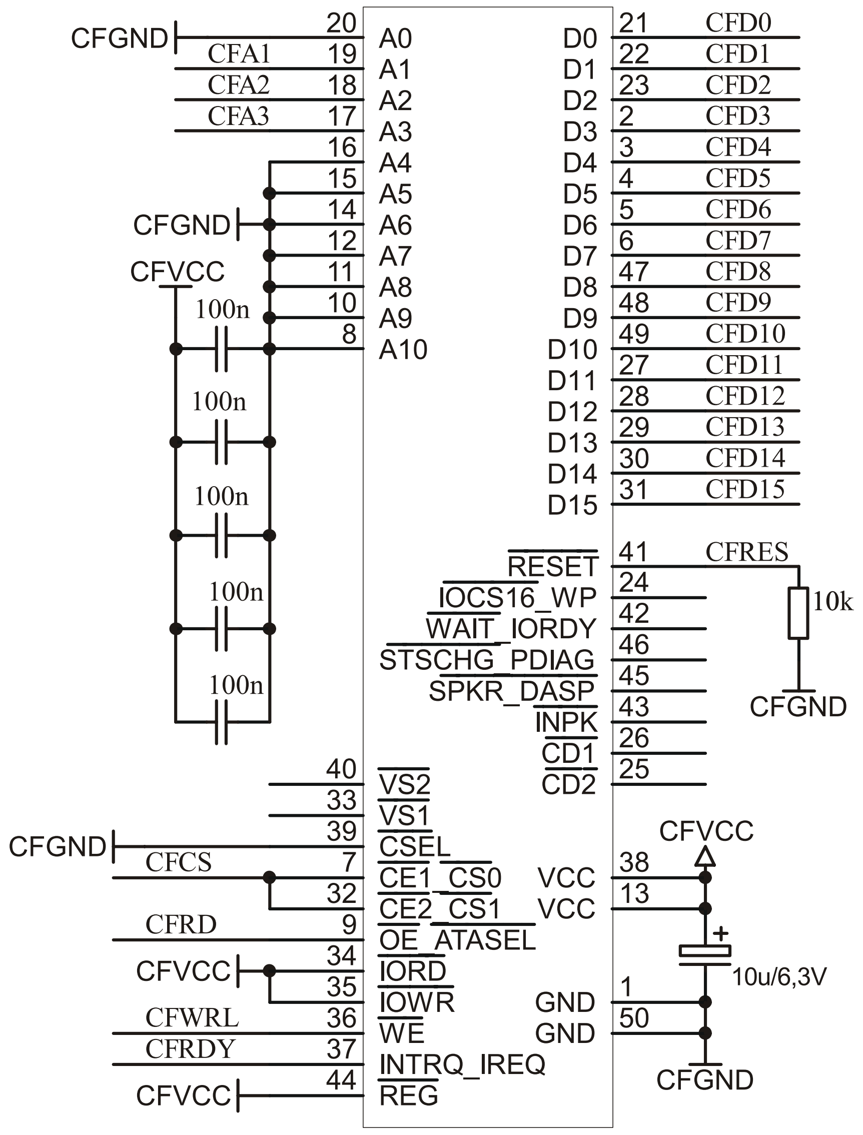
Signal name	Dir	Pin	Description
A10 - A0	I	8, 10, 11, 12, 14, 15, 16, 17, 18, 19, 20	These address lines along with the -REG signal are used to select the following: the I/O port address registers within the CompactFlash Storage Card or CF+ Card, the memory mapped port address registers within the CompactFlash Storage Card or CF+ Card, a byte in the card's information structure and its configuration control and status registers.
BVD1	I/O	46	This signal is asserted high, as BVD1 is not supported.
BVD2	I/O	45	This signal is asserted high, as BVD2 is not supported.
$\overline{CD1}$, $\overline{CD2}$	O	26, 25	These Card Detect pins are connected to ground on the CompactFlash Storage Card or CF+ Card. They are used by the host to determine that the CompactFlash Storage Card or CF+ Card is fully inserted into its socket.
$\overline{CE1}$, $\overline{CE2}$	I	7, 32	These input signals are used both to select the card and to indicate to the card whether a byte or a word operation is being performed. -CE2 always accesses the odd byte of the word. We recommend connecting these pins together.
CSEL	I	39	This signal is not used for this mode, but should be grounded by the host.
D15 - D00	I/O	27 - 31 47 - 49 2 - 6 21 - 23	These lines carry the Data, Commands and Status information between the host and the controller. D00 is the LSB of the Even Byte of the Word. D08 is the LSB of the Odd Byte of the Word.
GND	--	1, 5	Ground.
INPACK	O	43	This signal is not used in this mode.
IORD	I	34	This signal is not used in this mode.
IOWR	I	35	This signal is not used in this mode.
$\overline{OE}$ (ATA SEL)	I	9	This is an Output Enable strobe generated by the host interface. It is used to read data from the CompactFlash Storage Card or CF+ Card in Memory Mode and to read the CIS and configuration registers.

Table 5.73: Pin functions in memory card mode

Signal name	Dir	Pin	Description
READY	O	37	In Memory Mode, this signal is set high when the CompactFlash Storage Card or CF+ Card is ready to accept a new data transfer operation and is held low when the card is busy. At power up and at Reset, the READY signal is held low (busy) until the CompactFlash Storage Card or CF+ Card has completed its power up or reset function. No access of any type should be made to the CompactFlash Storage Card or CF+ Card during this time.Note, however, that when a card is powered up and used with +RESET continuously disconnected or asserted, the reset function of this pin is disabled and consequently the continuous assertion of +RESET will not cause the READY signal to remain continuously in the busy state.
REG	I	44	This signal is used during Memory Cycles to distinguish between Common Memory and Register (Attribute) Memory accesses. High for Common Memory, Low for Attribute Memory. To use it with µC/FS, this signal should be high.
RESET	I	41	When the pin is high, this signal Resets the CompactFlash Storage Card or CF+ Card. The CompactFlash Storage Card or CF+ Card is reset only at power up if this pin is left high or open from power-up.
VCC	--	13, 38	+5 V, +3.3 V power.
$\overline{\text{VS1}}$, $\overline{\text{VS2}}$	O	33, 4	Voltage Sense Signals. -VS1 is grounded so that the CompactFlash Storage Card or CF+ Card CIS can be read at 3.3 volts and -VS2 is reserved by PCMCIA for a secondary voltage.
WAIT	O	42	The -WAIT signal is driven low by the CompactFlash Storage Card or CF+ Card to signal the host to delay completion of a memory or I/O cycle that is in progress.
WE	I	36	This is a signal driven by the host and used for strobing memory write data to the registers of the CompactFlash Storage Card or CF+ Card when the card is configured in the memory interface mode.
WP	O	24	The CompactFlash Storage Card or CF+ Card does not have a write protect switch. This signal is held low after the completion of the reset initialization sequence.

Table 5.73: Pin functions in memory card mode(Continued)

Sample block schematic



5.5.2.2 IDE (ATA) Drives

Just like Compact Flash cards, ATA drives have a built-in controller to drive and control the mechanical hardware in a drive. Actually there are two types of connecting ATA drives. 5.25 and 3.5 inch drives are using a 40 pin male interface to connect to an IDE controller. 2.5 and 1.8 inch drives, mostly used in Notebooks and embedded systems, have a 50 pin male interface.

Modes of operation (interface modes)

ATA drives can operate in a variety of different modes:

- PIO (Programmed I/O)
- Multiword DMA
- Ultra DMA

Supported modes of operation (interface modes)

Currently, only PIO mode through TRUE IDE is supported.

ATA drives: True IDE mode pin functions

Refer to “True IDE mode pin functions” on page 225 for information.

ATA drives: Hardware interfaces

Signal	Pin	Pin	Signal
RESETÐ	1	2	Ground
DD7	3	4	DD8
DD6	5	6	DD9
DD5	7	8	DD10
DD4	9	10	DD11
DD3	11	12	DD12
DD2	13	14	DD13
DD1	15	16	DD14
DD0	17	18	DD15
Ground	19	20	key (no pin)
DMARQ	21	22	Ground
DIOWÐ	23	24	Ground
DIORÐ	25	26	Ground
IORDY	27	28	SPSYNC:CSEL
DMACKÐ	29	30	Ground
INTRQ	31	32	IOCS16Ð
DA1	33	34	PDIAGÐ
DA0	35	36	DA2
CS1FXÐ	37	38	CS3FXÐ
DASPÐ	39	40	Ground

Signal / use	Pin	Pin	Signal / use
master/slave jumper	A	B	master/slave jumper
master/slave jumper	C	D	master/slave jumper
no pin			no pin
RESET–	1	2	Ground
DD7	3	4	DD8
DD6	5	6	DD9
DD5	7	8	DD10
DD4	9	10	DD11
DD3	11	12	DD12
DD2	13	14	DD13
DD1	15	16	DD14
DD0	17	18	DD15
Ground	19	20	key (no pin)
DMARQ	21	22	Ground
DIOW–	23	24	Ground
DIOR–	25	26	Ground
IORDY	27	28	SPSYNC:CSEL
DMACK–	29	30	Ground
INTRQ	31	32	IOCS16–
DA1	33	34	PDIAG–
DA0	35	36	DA2
CS1FX–	37	38	CS3FX–
DASP–	39	40	Ground
+5V (logic)	41	42	+5V (motor)
+Ground	43	44	Type

5.5.3 Fail-safe operation

Unexpected Reset

The data will be preserved.

Power failure

Power failure can be critical: If the card does not have sufficient time to complete a write operation, data may be lost. Countermeasures: make sure the power supply for the card drops slowly.

5.5.4 Wear-leveling

CompactFlash card are controlled by an internal controller, this controller also handles wear leveling. Therefore, the driver does not need to handle wear-leveling.

5.5.5 Configuring the driver

5.5.5.1 Adding the driver to µC/FS

To add the driver, use `FS_AddDevice()` with the driver label `FS_IDE_Driver`. This function has to be called from within `FS_X_AddDevices()`. Refer to “FS_X_AddDevices()” on page 262 for more information.

Example

```
FS_AddDevice(&FS_IDE_Driver);
```

5.5.5.2 FS_IDE_Configure()

Description

Configures the IDE/CF drive. This function has to be called from `FS_X_AddDevices()`. `FS_IDE_Configure()` can be called before or after adding the device driver to the file system. Refer to "FS_X_AddDevices()" on page 262 for more information.

Prototype

```
void FS_IDE_Configure(U8  Unit, U8  IsSlave);
```

Parameter	Description
`Unit`	Unit number (0…N).
`ISSlave`	Specifies whether the unit is connected

Table 5.74: FS_IDE_Configure() parameter list

Additional information

This function only needs to be called when the device does not use the default IDE master/slave configuration. By default, all even units (0,2,4 ...) are master, all odd units are slave (1, 3, 5 ...).

Example

Configure 2 different IDE/CF devices:

```
void FS_X_AddDevices(void) {
  //
  //  Add 2 instances of the IDE driver
  //
  FS_AddDevice(&FS_IDE_Driver);
  FS_AddDevice(&FS_IDE_Driver);
  //
  //  Set the first unit as MASTER
  //
  FS_IDE_Configure(0, 0);
  //
  //  Set the second unit as MASTER
  //
  FS_IDE_Configure(1, 0);
}
```

5.5.6 Hardware functions

Routine	Explanation
Control line function	
`FS_IDE_HW_Reset()`	Resets the bus interface.
`FS_IDE_HW_Delay400ns()`	Waits 400ns.
`FS_IDE_HW_IsPresent()`	Checks if a device is present.
ATA I/O register access functions	
`FS_IDE_HW_ReadReg()`	Reads an IDE register. Data from the IDE register are read 16-bit wide.
`FS_IDE_HW_WriteReg()`	Write an IDE register. Data to the IDE register are written 16-bit wide.
`FS_IDE_HW_ReadData()`	Reads data from the IDE data register.
`FS_IDE_HW_WriteData()`	Writes data to the IDE data register.

Table 5.75: CompactFlash / IDE device driver functions

5.5.6.1 FS_IDE_HW_Reset()

Description

Resets the bus interface.

Prototype

```
void FS_IDE_HW_Reset (U8 Unit);
```

Parameter	Meaning
`Unit`	Unit number (0...N).

Table 5.76: FS_IDE_HW_Reset() parameter list

Additional Information

This function is called, when the driver detects a new media is present. For ATA HD drives, there is no action required and this function can be empty.

Example

```
void FS_IDE_HW_X_Reset(U8 Unit) {
  FS_USE_PARA(Unit);
}
```

5.5.6.2 FS_IDE_HW_Delay400ns()

Description

Waits 400ns.

Prototype

```
void FS_IDE_HW_Delay400ns (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0…N).

Table 5.77: FS_IDE_HW_Delay400ns() parameter list

Additional Information

`FS_IDE_HW_X_Delay400ns()` is always called when a command is sent or parameters are set in the IDE/CF drive. The integrated logic may need a delay of 400ns.
When using slow IDE/CF drives with fast processors this function should guarantee that a delay of 400ns is kept.
However this function may be empty if you intend to use fast drives (Modern CF-Cards and IDE drives are faster than 400ns when executing commands.)

Example

```
void FS_IDE_HW_X_Delay400ns(U8 Unit) {
  FS_USE_PARA(Unit);
}
```

5.5.6.3 FS_IDE_HW_IsPresent()

Description

Checks if the device is connected.

Prototype

```
U8 FS_IDE_HW_IsPresent (U8 Unit);
```

Parameter	Meaning
Unit	Unit number (0...N).

Table 5.78: FS_IDE_HW_IsPresent() parameter list

Return value

== 1: Device is connected.
== 0: Device is not connected.

Example

```
int FS_IDE_HW_IsPresent(U8 Unit) {
  FS_USE_PARA(Unit);
  return 1;
}
```

5.5.6.4 FS_IDE_HW_ReadReg()

Description

Reads an IDE register. Data from the IDE register are read 16-bit wide.

Prototype

```
U16 FS_IDE_HW_ReadReg (U8       Unit,
                       unsigned AddrOff);
```

Parameter	Meaning
`Unit`	Unit number (0...N).
`AddrOff`	Address offset that specifies which IDE register should be read.

Table 5.79: FS_IDE_HW_ReadReg() parameter list

Return value

Data read from the IDE register.

Example

```
U16  FS_IDE_HW_ReadReg(U8 Unit, unsigned AddrOff) {
  volatile U16 * pIdeReg;

  FS_USE_PARA(Unit);
  pIdeReg = _Getp(AddrOff);
  return *pIdeReg;
}
```

5.5.6.5 FS_IDE_HW_WriteReg()

Description

Writes an IDE register. Data to the IDE register are written 16-bit wide.

Prototype

```
void FS_IDE_HW_WriteReg (U8       Unit,
                         unsigned AddrOff,
                         U16      Data);
```

Parameter	Meaning
Unit	Unit number (0...N).
AddrOff	Address offset that specifies which IDE register should be written.
Data	Value that should be written to the register.

Table 5.80: FS_IDE_HW_WriteReg() parameter list

Example

```
void FS_IDE_HW_WriteReg(U8 Unit, unsigned AddrOff, U16 Data) {
  volatile U16 * pIdeReg;

  FS_USE_PARA(Unit);
  pIdeReg = _Getp(AddrOff);
  *pIdeReg = Data;
}
```

5.5.6.6 FS_IDE_HW_ReadData()

Description

Reads data from the IDE data register.

Prototype

```
void FS_IDE_HW_ReadData (U8         Unit,
                         U16        pData,
                         unsigned   NumBytes);
```

Parameter	Meaning
Unit	Unit number (0...N).
pData	Pointer to a read buffer.
NumBytes	Number of bytes that should be read.

Table 5.81: FS_IDE_HW_ReadData() parameter list

Example

```
void FS_IDE_HW_ReadData(U8 Unit, U8 * pData, unsigned NumBytes) {
  unsigned       NumItems;
  volatile U16 * pIdeReg;
  U16          * pData16;

  pIdeReg = _Getp(AddrOff);
  NumItems = NumBytes >> 1;
  pData16 = (U16 *)pData;
  do {
    *pData16++ = *pIdeReg;
  }  while (--NumItems);
}
```

5.5.6.7 FS_IDE_HW_WriteData()

Description

Writes data to the IDE data register.

Prototype

```
void FS_IDE_HW_WriteData (U8        Unit,
                          U16       Data,
                          unsigned  NumBytes);
```

Parameter	Meaning
Unit	Unit number (0...N).
pData	Pointer to a buffer of data which should be written.
NumBytes	Number of bytes that should be read.

Table 5.82: FS_IDE_HW_WriteData() parameter list

Example

```
void FS_IDE_HW_WriteData(U8 Unit, const U8 * pData, unsigned NumBytes) {
  unsigned       NumItems;
  volatile U16 * pIdeReg;
  U16          * pData16;

  pIdeReg = _Getp(AddrOff);
  NumItems = NumBytes >> 1;
  pData16 = (U16 *)pData;
  do {
    *pIdeReg = *pData16++;
  }  while (--NumItems);
}
```

5.5.7 Additional information

The µC/FS's generic CompactFlash & IDE device driver can be used to access most ATA HD drives or CompactFlash storage cards also known as CF using true IDE or Memory card mode. For details on CompactFlash cards, check the specification, which is available at:

http://www.compactflash.org/

Information about the AT Attachment interface can be found at the Technical Committee T13, who is responsible for the ATA standard:

http://www.t13.org/

5.5.8 Resource usage

5.5.8.1 ROM usage

The ROM usage depends on the compiler options, the compiler version, and the used CPU. The memory requirements of the IDE/CF driver displayed in the table have been measured on a system as follows: ARM7, IAR Embedded Workbench V4.41A, Thumb mode, Size optimization.

Module	ROM [Kbytes]
µC/FS IDE/CF driver	1.6

5.5.8.2 Static RAM usage

Static RAM usage is the amount of RAM required by the driver for variables inside the driver. The number of bytes can be seen in a list file.

Static RAM usage of the IDE/CF driver: 24 bytes.

5.5.9 FAQs

None.

5.6 NOR flash driver

µC/FS supports the use of NOR flashes. An optional driver for NOR flashes is available. The NOR flash driver can work with almost any NOR flash and is extremly efficient.

This section first describes which devices are supported and will afterwards describe the configuration and the additional information functions of µC/FS's NOR flash driver.

5.6.1 Supported hardware

The NOR flash driver can be used with almost any NOR flash. This includes NOR flashes with 1x8-bit and 1x16-bit parallel interfaces, as well as 2x16-bit interfaces in parallel, as well as serial NOR flashes.

Requirements

To be more precise, any NOR flash which fulfills the following requirements:

- Minimum of 2 physical sectors. At least 2 sectors need to be identical in size.
- Physical sectors need to be at least 2048 bytes each.
- Physical sectors do not need to be uniform
  (for example, 8 * 8 Kbytes + 3 * 64 Kbytes is permitted).
- Flash needs to be re-writeable without erase: The same location can be written to multiple times without erase, as long as only 1-bits are converted to 0-bits.
- Erase clears all bits in a physical sector to 1.

Physical layer

The driver requires a physical layer for the flash device.

The following physical layers are available:

- FS_NOR_PHY_CFI_1x16 - CFI compliant parallel NOR flash with 1x16-bit interface
- FS_NOR_PHY_CFI_2x16 - CFI compliant parallel NOR flash with 2x16-bit interface
- FS_NOR_PHY_SERIALFLASH_M25P - Serial flash (ST M25Pxx family)
- Physical layer template

Common flash interface (CFI)

The NOR flash driver can be used with any CFI-compliant 16-bit chip. The Common Flash Memory Interface (CFI) is an open specification which may be implemented freely by flash memory vendors in their devices. It was developed jointly by Intel, AMD, Sharp, and Fujitsu.

The idea behind CFI was the interchangeability of current and future flash memory devices offered by different vendors. If you use only CFI compliant flash memory chips, you are able to use one driver for different flash products by reading identifying information out of the flash chip itself.

The identifying information for the device, such as memory size, byte/word configuration, block configuration, necessary voltages, and timing information, is stored directly on the chip.

5.6.1.1 Tested and compatible NOR flashes

In general, the driver supports almost all serial and parallel NOR flashes which fulfill the listed requirements. This includes NOR flashes with 1x8-bit, 1x16-bit and 2x16-bit interfaces.

The table below shows the serial NOR flashes that have been tested or are compatible with a tested device:

Manufacturer	Device	Size
ST Microelectronics	M25P40 M25P80 M25P16 M25P32 M25P128	4 Mbytes (512 Kbytes x 8) 8 Mbytes (1Mbytes x 8) 16 Mbytes (2Mbytes x 8) 32 Mbytes (4Mbytes x 8) 128 Mbytes (16Mbytes x 8)

Table 5.83: List of supported serial NOR flashes

The table below shows the parallel NOR flashes that have been tested or are compatible with a tested device:

Manufacturer	Device	Size [Bits]
Intel	Intel 28FxxxP30 Intel 28FxxxP33	64 Mbytes - 1 Gbytes 64 Mbytes - 512 Mbytes
ST-Microelectronics	M28W160 M28W320 M28W640 M29F080 M29W160 M29W320 M29W640 M58LW064	16 Mbytes (1 Mbytes x 16) 32 Mbytes (2 Mbytes x 16) 64 Mbytes (4 Mbytes x 16) 8 Mbytes (1 Mbytes x 8) 16 Mbytes (2 Mbytes x 8 or 1 Mbytes x 16) 32 Mbytes (4 Mbytes x 8 or 2 Mbytes x 16) 64 Mbytes (8 Mbytes x 8 or 4 Mbytes x 16) 64 Mbytes (8 Mbytes x 8, 4Mbytes x 16)
Micron	MT28F128 MT28F256 MT28F320 MT28F640	128 Mbytes 256 Mbytes 32 Mbytes 64 Mbytes

Table 5.84: List of supported serial NOR flashes

Support for devices not available in this list

Most other NOR flash devices are compatible with one of the supported devices. Thus the driver can be used with these devices or may only need a little modification, which can be easily done. Get in touch with us, if you have questions about support for devices not in this list.

5.6.2 Theory of operation

Differentiating between "logical sectors" or "blocks" and "physical sectors" is very essential to understand this section. A logical sector/block is the base unit of any file system, its usual size is 512 bytes. A physical sector is an array of bytes on the flash chip that are erased together (typically between 2 Kbytes - 128 Kbytes). The flash chip driver is an abstraction layer between these two types of sectors.

Every time a logical sector is being updated, it is marked as invalid and the new content of this sector is written into another area of the flash. The physical address and the order of physical sectors can change with every write access. Hence, there cannot exist a direct relation between the sector number and its physical location.

The flash driver manages the logical sector numbers by writing it into special headers. It does not matter to the upper layer were the logical sector is stored or how much flash memory is used as a buffer. All logical sectors (starting with Sector #0) always exist and are always available for user access.

Using the same NOR flash for code and data

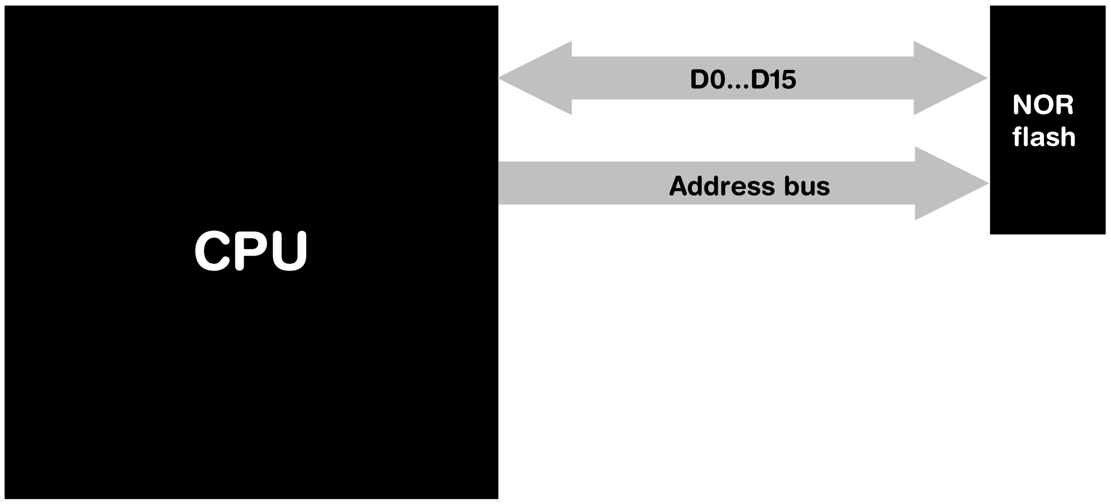
Most NOR flashes cannot be read out during a program, erase or identify operation. This means that code cannot be read from the NOR flash during a program or erase operation. If code which resides in the same NOR flash used for data storage is executed during program or erase, a program crash is almost certain.

There are multiple options to solve this:

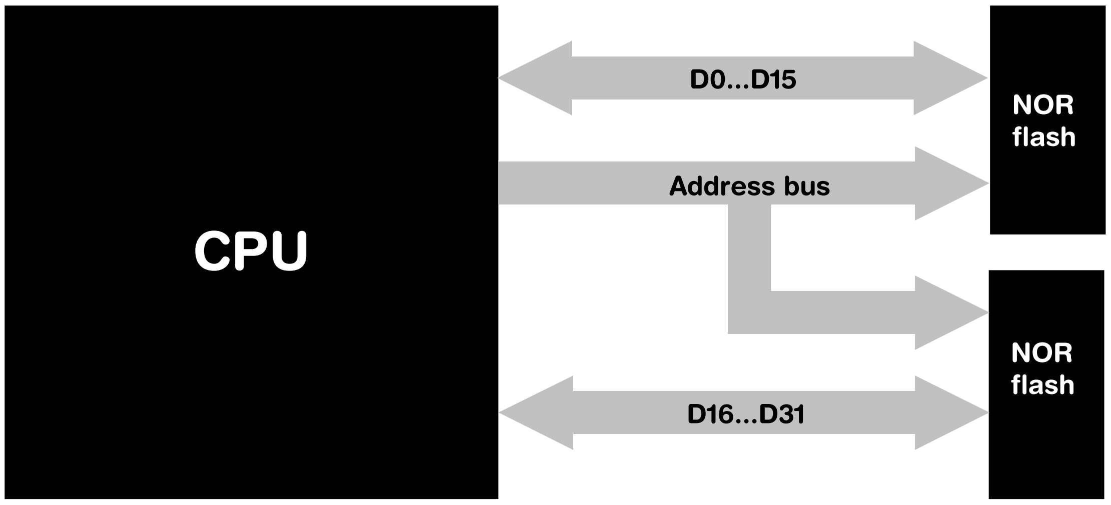
1. Use multiple NOR flashes. Use one flash for code and one for data.
2. Use a NOR flash with multiple banks, which allows reading Bank A while Bank B is being programmed.
3. Make sure the hardware routines which program, erase or identify the NOR flash are located in RAM and interrupts are disabled.

Physical interfaces

A device can consist of a single or two identical CFI compliant flash interfaces with a 16-bit interface. The most common is a CFI compliant NOR flash chip with a 16-bit interface.



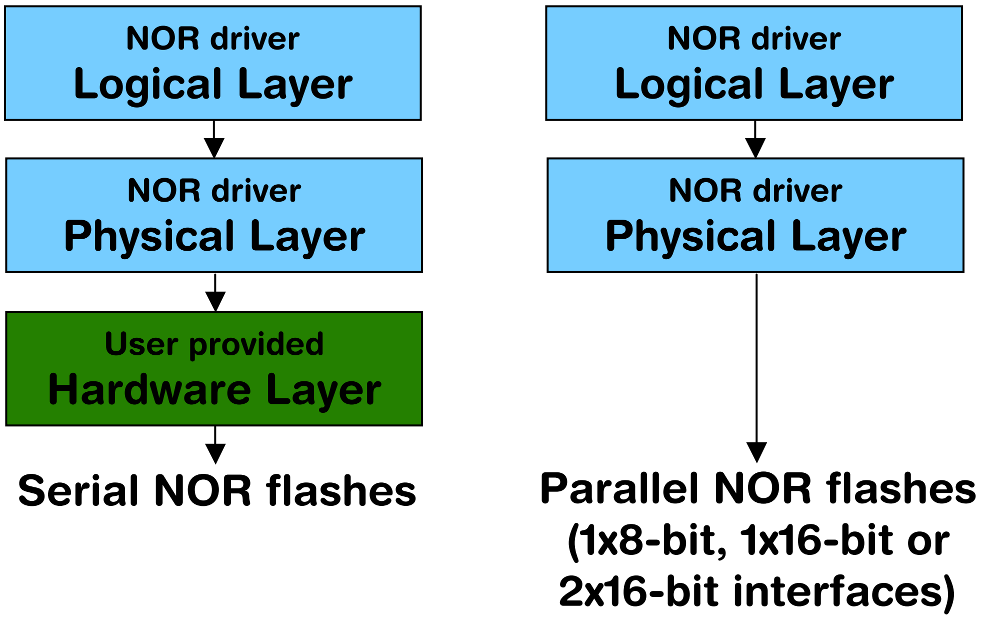
Beside this solution, µC/FS supports two CFI compliant NOR flash chip with a 16 bit interface which are connected to the same data bus.



The µC/FS NOR flash driver supports both options.

5.6.2.1 Software structure

The NOR flash driver is divided into different layers, which are shown in the illustration below.



It is possible to use the NOR flash driver also with serial NOR flashes. Only the hardware layer needs to be ported. Normally no changes to the physical layer are required. If the physical layer needs to be adapted, a template is available.

5.6.3 Fail-safe operation

The µC/FS NOR driver is fail-safe. That means that the driver makes only atomic actions and takes the responsibility that the data managed by the file system is always valid. In case of power loss or power reset during a write operation it is always assured that only valid data is stored in the flash. If the power loss interrupts the write operation, the old data will be kept and not corrupted.

5.6.4 Wear leveling

Wear leveling is supported by the driver. Wear leveling makes sure that the number of erase cycles remains approximately equal for each sector. Maximum erase count difference is set to 5. This value specifies a maximum difference of erase counts for different physical sectors before the wear leveling uses the sector with the lowest erase count.

5.6.5 Configuring the driver

5.6.5.1 Adding the driver to µC/FS

To add the driver, use `FS_AddDevice()` with the driver label `FS_NOR_Driver`. This function has to be called from `FS_X_AddDevices()`. Refer to "FS_X_AddDevices()" on page 262 for more information.

Example

```
FS_AddDevice(&FS_NOR_Driver);
```

5.6.5.2 FS_NOR_Configure()

Description

Configures the NOR flash drive. This function has to be called from `FS_X_AddDevices()` after adding the device driver to file system. Refer to "FS_X_AddDevices()" on page 262 for more information.

Prototype

```
void FS_NOR_Configure(U8  Unit,
                      U32 BaseAddr,
                      U32 StartAddr,
                      U32 NumBytes);
```

Parameter	Description
`Unit`	Unit number (0...N).
`BaseAddr`	Base address of the NOR flash chip.
`StartAddr`	Start address of the NOR flash device.
`NumBytes`	Specifies the size of the NOR flash device in bytes.

Table 5.85: FS_NOR_Configure() parameter list

Additional information

If your consists of two identical CFI compliant NOR flash chips with 16 bit interface `FS_NOR_Configure()` configures both flash chips. Refer to "FS_NOR_SetPhyType()" on page 254 for more information about the different physical type of your device.

Example

Configure a single NOR flash chip:

```
void FS_X_AddDevices(void) {
  //
  //  Add driver
  //
  FS_AddDevice(&FS_NOR_Driver);
  //
  //  Set physical type, single CFI compliant NOR flash chips with 16 bit interface
  //
  FS_NOR_SetPhyType(0, &FS_NOR_PHY_CFI_1x16);
  //
  //  Configure a single NOR flash interface (256 Mbytes)
  //
  FS_NOR_Configure(0, 0x1000000, 0x1000000, 0x200000);
}
```

Configure two NOR flash chips:

```
void FS_X_AddDevices(void) {
  //
  //  Add driver
  //
  FS_AddDevice(&FS_NOR_Driver);
  //
  //  Set physical type, 2 identical CFI compliant NOR flash chips
  //  with 16 bit interface
  //
  FS_NOR_SetPhyType(0, &FS_NOR_PHY_CFI_2x16);

  //
  //  Configure two NOR flash interfaces (256 Mbytes each)
  //
  FS_NOR_Configure(0, 0x1000000, 0x1000000, 0x400000);
}
```

5.6.5.3 FS_NOR_SetPhyType()

Description

Sets the physical type of the device. The NOR flash driver comes with 2 different physical interfaces. The most common is a CFI compliant NOR flash chip with a 16 bit interface. A device can consist of a single or two identical CFI compliant flash interfaces with a 16 bit interface. Set `pPhyType` to `FS_NOR_PHY_CFI_1x16` if you use a single NOR flash chip. If your device consists of two identical NOR flash chips, set `pPhyType` to `FS_NOR_PHY_CFI_2x16`.

This function has to be called from within `FS_X_AddDevices()` after adding the device driver to file system. Refer to "FS_X_AddDevices()" on page 262 for more information.

Prototype

```
void FS_NOR_SetPhyType(U8 Unit, const FS_NAND_PHY_TYPE * pPhyType);
```

Parameter	Meaning
`Unit`	Unit number (0...N).
`pPhyType`	Pointer to physical type.

Table 5.86: FS_NOR_SetPhyType() parameter list

Permitted values for parameter `pPhyType`	
`FS_NOR_PHY_CFI_1x16`	One CFI compliant NOR flash chip with 16 bit interface.
`FS_NOR_PHY_CFI_2x16`	Two CFI compliant NOR flash chip with 16 bit interfaces.

Additional information

If you want to access special flash devices (for example, the internal NOR flash of a microcontroller), you can define your own physical type. Use the supplied template `NOR_Phy_Template.c` for the implementation. The template is located in the `\Sample\Driver\NOR\` directory.

Note: Most NOR flashes cannot be read out during a program, erase or identify operation. This means that code cannot be read from the NOR flash during a program or erase operation. If code which resides in the same NOR flash used for data storage is executed during program or erase, a program crash is almost certain. To avoid this, you have to make sure that routines which program, erase or identify are located in RAM and interrupts are disabled. The responsibility therefor is on user side.

Example

Refer to “FS_NOR_Configure()” on page 253 for an example of usage.

5.6.6 Hardware functions

The NOR flash driver for CFI complant chips does not need any hardware function.

5.6.7 Additional Information

Low-level format

Before using the NAND flash as storage device. A low-level format has to be performed. Refer to “FS_FormatLow()” on page 83 and “FS_FormatLLIfRequired()” on page 82 for detailed information about low-level formatting.

5.6.7.1 Further reading

For more technical details about CFI compliant flash memory, check the documents and specifications that are available free of charge:

- *Common Flash Interface (CFI) and Command Sets*
  Intel - Application Note 646 - April 2000
- *Common Flash Memory Interface Specification*
  AMD - Revision 2.0 - December 1, 2001

5.6.8 Additional driver functions

5.6.8.1 FS_NOR_GetDiskInfo()

Description

Returns information about the flash disk.

Prototype

```
void FS_NOR_GetDiskInfo(U8 Unit, FS_NOR_DISK_INFO * pDiskInfo);
```

Parameter	Description
`Unit`	Unit number (0...N).
`pDiskInfo`	Pointer to a structure of type `FS_NOR_DISK_INFO`.

Table 5.87: FS_NOR_GetDiskInfo() parameter list

Additional information

Refer to “Structure FS_NOR_DISK_INFO” on page 259 for more information about the structure elements.

5.6.8.2 FS_NOR_GetSectorInfo()

Description

Returns info about a particular physical physical sector.

Prototype

```
void FS_NOR_GetSectorInfo(U8                    Unit,
                          U32                   PhysSectorIndex,
                          FS_NOR_SECTOR_INFO * pSectorInfo);
```

Parameter	Description
`Unit`	Unit number (0...N).
`PhysSectorIndex`	Index of physical sector.
`pDiskInfo`	Pointer to a structure of type `FS_NOR_DISK_INFO`.

Table 5.88: FS_NOR_GetSectorInfo() parameter list

Additional information

Refer to “Structure FS_NOR_SECTOR_INFO” on page 260 for more information about the structure elements.

Example

```
/*********************************************************************
*
*       ShowDiskInfo
*
*/
void ShowDiskInfo(FS_NOR_DISK_INFO* pDiskInfo) {
  char acBuffer[80];

  FS_X_Log("Disk Info: \n");
  FS_NOR_GetDiskInfo(0, pDiskInfo);
  sprintf(acBuffer," Physical sectors: %d\n"
                   " Logical sectors : %d\n"
                       " Used sectors: %d\n", pDiskInfo->NumPhysSectors, pDiskInfo-
>NumLogSectors, pDiskInfo->NumUsedSectors);
  FS_X_Log(acBuffer);
}
/*********************************************************************
*
*       ShowSectorInfo
*/
void ShowSectorInfo(FS_NOR_SECTOR_INFO* pSecInfo, U32 PhysSectorIndex) {
  char acBuffer[400];

  FS_X_Log("Sector Info: \n");
  FS_NOR_GetSectorInfo(0, PhysSectorIndex, pSecInfo);
  sprintf(acBuffer," Physical sector No.     : %d\n"
                   " Offset                  : %d\n"
                   " Size                    : %d\n"
                   " Erase Count             : %d\n"
                   " Used logical sectors    : %d\n"
                   " Free logical sectors    : %d\n"
                   " Erasable logical sectors: %d\n", PhysSectorIndex,
                                                      pSecInfo->Off,
                                                      pSecInfo->Size,
                                                      pSecInfo->EraseCnt,
                                                      pSecInfo->NumUsedSectors,
                                                      pSecInfo->NumFreeSectors,
                                                     pSecInfo->NumEraseableSectors);
  FS_X_Log(acBuffer);
}

/*********************************************************************
```

```
*
*       MainTask
*/
void MainTask(void) {
  U32                 i, j;
  char                ac[0x400];
  FS_NOR_DISK_INFO    DiskInfo;
  FS_NOR_SECTOR_INFO  SecInfo;

  FS_FILE * pFile;
  FS_Init();
  FS_FormatLLIfRequired("");
  for(i = 0; i < strlen(ac); i++) {
    ac[i] = 'A';
  }
  //
  // Check if volume needs to be high-level formatted.
  //
  if (FS_IsHLFormatted("") == 0) {
    printf("High level formatting\n");
    FS_Format("", NULL);
  }
  ShowDiskInfo(&DiskInfo);
  for (i = 0;  i < 1000; i++) {
    pFile = FS_FOpen("Test.txt","w");
    if(pFile != 0) {
      FS_Write(pFile, &ac, strlen(ac));
      FS_FClose(pFile);
      printf("Loop cycle: %d\n", i);
      for(j = 0; j < DiskInfo.NumPhysSectors; j++) {
        ShowSectorInfo(&SecInfo, j);
      }
    }
  }
  while(1);
}
```

5.6.8.3 Structure FS_NOR_DISK_INFO

Description

The `FS_NOR_SECTOR_INFO` structure contains physical and logical sector information.

Prototype

```
typedef struct {
  U32 NumPhysSectors;
  U32 NumLogSectors;
  U32 NumUsedSectors;   /* Number of used logical sectors */
} FS_NOR_DISK_INFO;
```

Members	Description
`NumPhysSectors`	Number physical sectors of the chip.
`NumLogSectors`	Number of logical sectors of the chip.
`NumUsedSectors`	Number of used sectors of the chip.

Table 5.89: FS_NOR_DISK_INFO - list of structure elements

5.6.8.4 Structure FS_NOR_SECTOR_INFO

Description

The `FS_NOR_SECTOR_INFO` structure contains physical and logical sector information.

Prototype

```
typedef struct {
  U32 Off;
  U32 Size;
  U32 EraseCnt;
  U16 NumUsedSectors;
  U16 NumFreeSectors;
  U16 NumEraseableSectors;
} FS_NOR_SECTOR_INFO;
```

Members	Description
`Off`	Offset of the physical sector.
`Size`	Size of the physical sector.
`EraseCnt`	Erase count of sector.
`NumUsedSectors`	Number of used logical sector inside the physical sector.
`NumFreeSectors`	Number of free logical sector inside the physical sector.
`NumEraseableSectors`	Number of erasable logical sector inside the physical sector.

Table 5.90: FS_NOR_SECTOR_INFO - list of structure elements

5.7 *WinDrive* driver

The purpose of this driver is to run µC/FS for test and simulation purposes on a PC running Windows. Refer to the chapter "Getting started" on page 17 for a sample using the WinDrive driver.

5.7.1 Supported hardware

This driver is compatible with use any Windows logical driver on a Windows NT system.

Be aware, that Win9X is not supported, because it cannot access logical drives with "CreateFile".

5.7.2 Theory of operation

µC/FS supports in this version FAT and EFS file systems only. NTFS logical drives cannot be accessed by µC/FS. It can be used either to store/access files on a floppy disk or using an USB-Card reader for accessing flash cards. It works also on FAT formated hard disks or partitions.

Note: Do not use this driver on partitions containing important data. It is primarily meant to be used for evaluation purposes. Problems may occur if the program using µC/FS is debugged or terminated using the task manager.

5.7.3 Fail safe operation

Although not important since the driver is not designed to be used in an embedded device, the data is normally safe. Data safety is handled by the underlying operating system and hardware.

5.7.4 Wear leveling

The driver does not need wear leveling.

5.7.5 Configuring the driver

5.7.5.1 Adding the driver to µC/FS

To add the driver use `FS_AddDevice()` with the driver label `FS_WINDRIVE_Driver`. This function has to be called from within `FS_X_AddDevices()`. Refer to "FS_X_AddDevices()" on page 262 for more information.

Example

```
FS_AddDevice(&FS_WINDRIVE_Driver);
```

5.7.5.2 FS_Windrive_Configure()

Description

Configures a windows drive instance. This function has to be called from within `FS_X_AddDevices()` after adding an instance of the `Windrive` driver. Refer to "FS_X_AddDevices()" on page 262 for more information.

Prototype

```
void WINDRIVE_Configure(U8 Unit, const char * sDriveName);
```

Parameter	Description
`Unit`	Unit number (0...n).
`sDriveName`	Pointer to string which contains the windows drive name. For example: `"\\.\\a:"`

Table 5.91: FS_Windrive_Configure() parameter list

5.7.6 Hardware functions

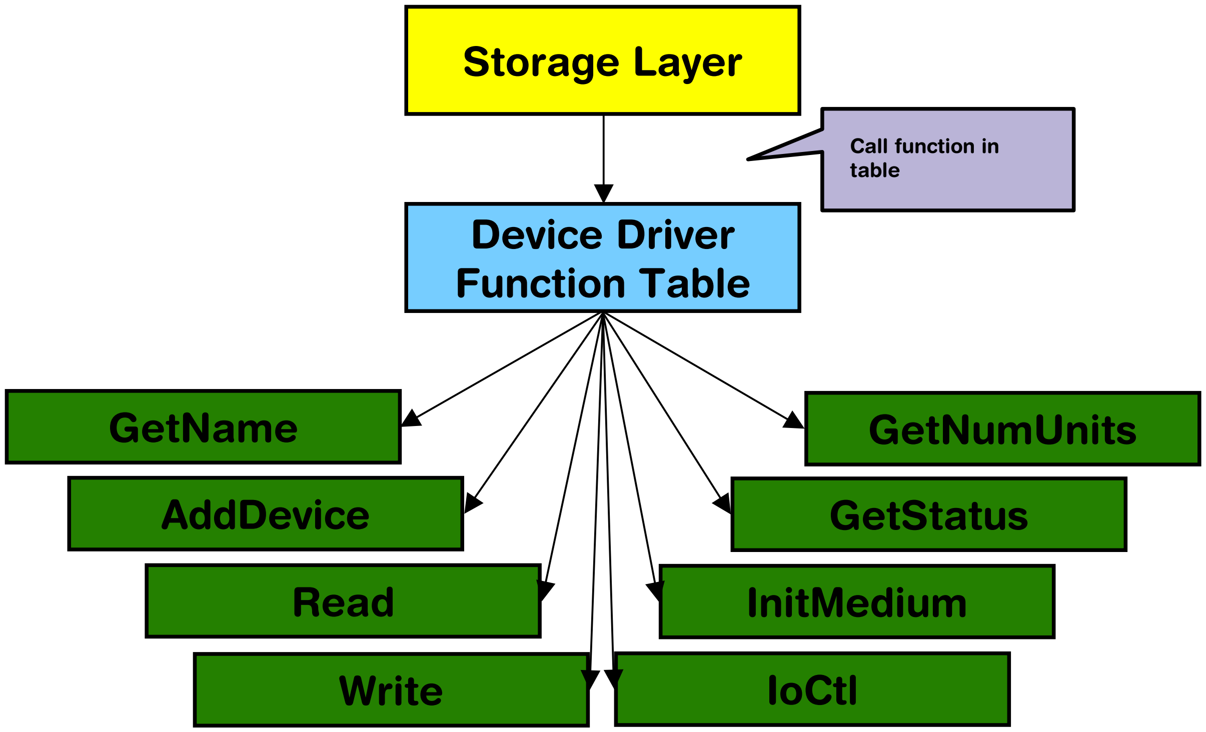
The WinDrive driver does not need any hardware functions.

5.7.7 Additional information

None.

5.8 Writing your own driver

If you are going to use µC/FS with your own hardware, you may have to write your own device driver. This section describes which functions are required and how to integrate your own device driver into µC/FS.



5.8.1 Device driver functions

This section provides descriptions of the device driver functions required by µC/FS. Note that the names used for these functions are not really relevant for µC/FS because the file system accesses them through a function table.

Routine	Explanation
`AddDevice()`	Adds a device to file system.
`GetName()`	Returns the name of the device.
`GetNumUnits()`	Returns the number of units.
`GetStatus()`	Returns the Status of the device.
`InitMedium()`	Initializes the device.
`IoCtl()`	Executes a special command on a device.
`Read()`	Reads data from a device.
`Write()`	Writes data to a device.

Table 5.92: Device driver functions

5.8.2 Device driver function table

µC/FS uses function tables to call the appropiate driver function for a device.

Data structure

```
typedef struct {
  const char *       (*pfGetName)      (U8             Unit);
  int                (*pfAddDevice)    (void);
  int                (*pfRead)         (U8             Unit,
                                        U32            SectorNo,
                                        void *         pBuffer,
                                        U32            NumSectors);
  int                (*pfWrite)        (U8             Unit,
                                        U32            SectorNo,
                                        const void *   pBuffer,
                                        U32            NumSectors,
                                        U8             RepeatSame);
  int                (*pfIoCtl)        (U8             Unit,
                                        I32            Cmd,
                                        I32            Aux,
                                        void *         pBuffer);
  int                (*pfInitMedium)   (U8             Unit);
  int                (*pfGetStatus)    (U8             Unit);
  int                (*pfGetNumUnits)  (void);
} FS_DEVICE_TYPE;
```

Elements of FS_DEVICE_TYPE

Element	Meaning
pfGetName	Pointer to a function that returns the name of the driver.
pfRead	Pointer to the device read sector function.
pfWrite	Pointer to the device write sector function.
pfIoCtl	Pointer to the device IoCtl function.
pfInitMedium	Pointer to the medium initialization function. (optional)
pfGetStatus	Pointer to the device status function.
pfGetNumUnits	Pointer to a function that returns the number of available devices.

Table 5.93: FS_DEVICE_TYPE - List of structure member variables

Example

```
/* sample implementation taken from the RAM device driver */

const FS_DEVICE_TYPE FS_RAMDISK_Driver = {
  _GetDriverName,
  _AddDevice,
  _Read,
  _Write,
  _IoCtl,
  NULL,
  _GetStatus,
  _GetNumUnits
};
```

5.8.3 Integrating a new driver

There is an empty skeleton driver called `generic` in the `Sample\Driver\DriverTemplate\` folder. This driver can be easily modified to get any block oriented storage device working with the file system.

To add the driver to µC/FS, `FS_AddDevice()` should be called from within `FS_X_AddDevices()` to mount the device driver to µC/FS before accessing the device or its units. Refer to “FS_X_AddDevices()” on page 262 for more information.

Chapter 6

µC/FS Configuration

µC/FS can be used without the need for changing any of the compile time flags. All compile time configuration flags are preconfigured with valid values, which matchs the requirements of the most applications. Device drivers can be added to runtime.

The default configuration of µC/FS can be changed via compile time flags which can be added to `FS_Conf.h`. `FS_Conf.h` is the main configuration file for the file system.

Every driver folder includes a configuration file (e.g. `ConfigRamDisk.c`) with implementations of runtime configuration functions explained in this chapter. The configuration files are a good start, to run µC/FS "out of the box".

6.1 Runtime configuration

Every driver folder includes a configuration file (e.g. `ConfigRamDisk.c`) with implementations of runtime configuration functions explained in this chapter. These functions can be customized.

6.1.1 Driver handling

`FS_AddDevices()` is called by the initialization of the file system from `FS_Init()`. This function should help to bundle the process of adding and configuring the driver.

6.1.1.1 FS_X_AddDevices()

Description

Helper function called by `FS_Init()` to add devices to the file system and configure them.

Prototype

```
void * FS_AddDevices (void);
```

Example

```
/*********************************************************************
*
*       FS_X_AddDevices
*/
void FS_X_AddDevices(void) {
  void  * pRamDisk;

  //
  // Allocate memory for the RAM disk
  //
  pRamDisk = FS_Alloc(RAMDISK_NUM_SECTORS * RAMDISK_BYTES_PER_SECTOR);
  //
  // Add driver
  //
  FS_AddDevice(&FS_RAMDISK_Driver);
  //
  // Configure driver
  //
  FS_RAMDISK_Configure(0, pRamDisk, RAMDISK_BYTES_PER_SECTOR, RAMDISK_NUM_SECTORS);
  //
  //  Check if volume is already high-level formatted
  //  otherwise high-level format the volume
  //
  if (FS_IsHLFormatted("") == 0) {
    FS_Format("", NULL);
  }
}
```

6.1.2 Memory allocation

6.1.2.1 FS_X_Alloc()

Description

Provides memory to the file system.

Prototype

```
void * FS_X_Alloc(U32 NumBytes);
```

Parameter	Description
NumBytes	Number of bytes that should be allocated.

Table 6.1: FS_X_Alloc() parameter list

Additional information

- Fragmentation:
  The file system allocates memory only in the configuration phase, not during normal operation, so that fragmentation should not occur.
- Failure:
  Since the memory is required for proper operation of the file system, this function does not return on failure. In case of a configuration problem where insufficient memory is available to the application, this is normally detected by the programmer in the debug phase.

Example

```
/*********************************************************************
*
*       FS_X_Alloc
*/
void * FS_X_Alloc(I32 NumBytes) {
  I32    NumItems;
  void * p;

  NumItems = (NumBytes + sizeof(U32) - 1) / sizeof(U32);
  if ((FS_ALLOC_NumItemsFree - NumItems) <= 0) {
    return NULL;
  }
  p               = _pNextBlock;
  _pNextBlock    += NumItems;
  FS_ALLOC_NumItemsFree -= NumItems;
  return p;
}
```

6.1.3 System configuration

6.1.3.1 FS_X_GetTimeDate()

Description

Returns the current time and date.

Prototype

```
U32 FS_X_OS_GetTimeDate(void);
```

Return value

Current time and date as U32 in a format suitable for the file system.

Additional Information

The format of the time is arranged as follows:
Bit 0-4: 2-second count (0-29)
Bit 5-10: Minutes (0-59)
Bit 11-15: Hours (0-23)
Bit 16-20: Day of month (1-31)
Bit 21-24: Month of year (1-12)
Bit 25-31: Number of years since1980 (0-127)

Example

```
U32 FS_X_GetTimeDate(void) {
  U32 r;
  U16 Sec, Min, Hour, Day, Month, Year;

  Sec   = FS_X_GET_SECOND();
  Min   = FS_X_GET_MINUTE();
  Hour  = FS_X_GET_HOUR();
  Day   = FS_X_GET_DAY();
  Month = FS_X_GET_MONTH();
  Year  = FS_X_GET_YEAR();

  r   = Sec / 2 + (Min << 5) + (Hour  << 11);
  r  |= (Day + (Month << 5) + (Year  << 9)) << 16;
  return r;
}
```

6.1.3.2 Logging functions

Logging is used in higher debug levels only. The typical target build does not use logging and does therefore not require any of the logging functions. For a release build without logging the functions may be eliminated from configuration file to save some space. (If the linker is not function aware and eliminates unreferenced functions automatically). Refer to “Debugging” on page 283 for further information about the different logging functions.

6.2 Compile time configuration

6.2.1 General file system configuration

Type	Macro	Default	Description
N	`FS_MAX_SECTOR_SIZE`	512	Defines the maximum sector size µC/FS should handle. This is required if you intend to use sector sizes greater than 512 bytes.
B	`FS_SUPPORT_FAT`	1	Defines if µC/FS should use the FAT file system layer. **Note:** FAT & EFS cannot be used simultaneously.
B	`FS_SUPPORT_EFS`	0	Defines if µC/FS should use the optional EFS file system layer.
B	`FS_SUPPORT_CACHE`	1	Determines whether `FS_AssignChache()` can be used. `FS_AssignChache()` allows run-time assigment of a cache. Refer to "FS_AssignCache()" on page 115 for further information. **Note:** `FS_AssignCache()` needs to be called to activate the cache functionality for a specific device.
B	`FS_MULTI_HANDLE_SAFE`	0	If you intend to open a file simultaneously for read/write, set this macro to 1.
S	`FS_DIRECTORY_DELIMITER`	'\\'	Defines the character that should be used to delimit directories.
	`FS_VERIFY_WRITE`	0	Verify every write sector operation (tests the driver and hardware). This switch should always off for production code. It is normally switched on only when investigating driver problems.

Table 6.2: General file system configuration macros

6.2.2 FAT configuration

The current version of µC/FS supports FAT12/FAT16/FAT32.

Type	Macro	Default	Description
B	`FS_FAT_FWRITE_UPDATE_DIR`	1	Setting this macro to 1, writing to a file always updates the directory entry of the file always after writing to the file (guarding the new data against unexpected RESET). *ON:* Directory entry is updated with every write operation.This is slower, but has the advantage that data written without close (in case of unexpected RESET) is not lost. *OFF:* Directory entry is updated when file is closed. This is faster, but has the disadvantage that any data written between open and unexpected RESET is lost.
B	`FS_FAT_SUPPORT_FAT32`	1	To enable support for FAT32 media, define this macro to 1.
B	`FS_FAT_OPTIMIZE_SEQ_CLUSTERS`	1	This macro enables a speed optimization when reading large continuous files sequentially. *ON:* Higher speed, Bigger code. *OFF:* Lower speed, Smaller code.

Table 6.3: FAT configuration macros

Type	Macro	Default	Description
B	FS_FAT_USE_FSINFO_SECTOR	1	When retrieving the free disk amount on large FAT32 volumes, this may take a long time, since the FAT table can extend to many MBytes. To improve this, this macro should be set to 1. This will enable the feature of using the FAT32 specific FSInfo sector. This sector stores the information of the free clusters that are available and the last known free cluster. *ON:* Higher speed, Bigger code. *OFF:* Lower speed, Smaller code.
B	FS_FAT_OPTIMIZE_DELETE	1	When deleting a large contiguous file on a FAT system, it may take some time to delete the FAT entries for the file. This macro set to 1 enables a sequence to accel-erate this operation. *ON:* Higher speed, Bigger code. *OFF:* Lower speed, Smaller code.
B	FS_FAT_SUPPORT_UTF8	0	When using the LFN package, the file/directory name is stored as Unicode string. This macros enables the support for accessing such files and directories, where characters in the file/directory name are others than the standard Latin characters such as Greek or Cyrillic. To open such a file the string should be UTF-8 encoded.

Table 6.3: FAT configuration macros

6.2.3 EFS configuration

Type	Macro	Default	Description
B	FS_EFS_FWRITE_UPDATE_DIR	0	Setting this macro to 1, writing to a file always updates the directory entry of the file always after writing to the file (guarding the new data against unexpected RESET). *ON:* Directory entry is updated with every write operation.This is slower, but has the advantage that data written without close (in case of unexpected RESET) is not lost. *OFF:* Directory entry is updated when file is closed. This is faster, but has the disadvantage that any data written between open and unexpected RESET is lost.
B	FS_EFS_CASE_SENSITIVE	0	If EFS file/directory operations should be case sensitive, define this macro to 1.

Table 6.4: EFS configuration macros

6.2.4 OS support

µC/FS can be used with operating systems. For no OS support at all, set all of them to 0. If you need support for an additional OS, you will have to provide functions described in the chapter “OS Integration” on page 277.

Type	Macro	Default	Description
B	FS_OS_LOCKING	0	Set this to 1 determines that an operating system should be used. When using an operating system, generally every file system operation is locked by a semaphore. When this macro is defined to 1 only one lock is used to lock each file system function (Coarse lock granularity). If FS_OS_LOCKING is defined to 2 the file system locks on every critical file system operation. (Fine lock granularity). Fine lock granularity requires more semaphores.

Table 6.5: Operating system support macros

Default setting of µC/FS is not configured for a multitasking environment.

6.2.5 Debugging

µC/FS can be configured to display debug information at higher debug levels to locate a problem (Error) or potential problem. To display information, µC/FS uses the logging routines (s. chapter “Debugging” on page 283).These routines can be blank, they are not required for the functionality of µC/FS. In a target system, they are typically not required in a release (production) build, since a production build typically uses a lower debug level.

Fatal errors are output using `FS_X_ErrorOut() if (FS_DEBUG_LEVEL >= 3)`
Warnings are output using `FS_X_Warn() if (FS_DEBUG_LEVEL >= 4)`
Messages are output using `FS_X_Log() if (FS_DEBUG_LEVEL >= 5)`

The file system calls this function depending on the configuration macro `FS_DEBUG_LEVEL`. The following table lists the permitted values for `FS_DEBUG_LEVEL`:

Value	Symbolic name	Explanation
0	FS_DEBUG_LEVEL_NOCHECK	No run-time checks are performed.
1	FS_DEBUG_LEVEL_CHECK_PARA	Parameter checks are performed to avoid crashes. (Default for target system)
2	FS_DEBUG_LEVEL_CHECK_ALL	Parameter checks and consistency checks are per-formed.
3	FS_DEBUG_LEVEL_LOG_ERRORS	Errors are recorded.
4	FS_DEBUG_LEVEL_LOG_WARNINGS	Errors and warnings are recorded.(Default for PC-simulation)
5	FS_DEBUG_LEVEL_LOG_ALL	Errors, warnings and messages are recorded.

Table 6.6: Debug level macros

6.2.6 Miscellaneous configurations

Type	Macro	Default	Description
B	FS_NO_CLIB	0	Setting this macro to 1, µC/FS does not use the standard C library functions (such as strcmp() etc.) of your compiler.

Table 6.7: Miscellaneous configuration macros

6.2.7 Sample configuration

The µC/FS configuration file `FS_Conf.h` is located in Micrium\Software\uC-FS\Config. µC/FS compiles and runs without any problem with the default settings. If you want to change the default configuration, insert the corresponding macros in the delivered FS_Conf.h.

```
/*
**********************************************************************
*                          Micrium, Inc.
*                      949 Crestview Circle
*                     Weston,  FL 33327-1848
*
*                              uC/FS
*
*              (c) Copyright 2001 - 2006, Micrium, Inc.
*                      All rights reserved.
*
***********************************************************************
**** µC/FS file system for embedded applications ****
µC/FS is protected by international copyright laws. Knowledge of the
source code may not be used to write a similar product. This file may
only be used in accordance with a license and should not be re-
distributed in any way. We appreciate your understanding and fairness.

----------------------------------------------------------------------
File        : FS_Conf.h
Purpose     : uC/FS compile-time configuration settings
---------------------------END-OF-HEADER------------------------------
*/
#ifndef _FS_CONF_H_
#define _FS_CONF_H_

#define FS_DEBUG_LEVEL        1
#define FS_MAX_SECTOR_SIZE  512

#endif  /* Avoid multiple inclusion */
```

Chapter 7

OS Integration

µC/FS is suitable for any multithreaded environment. To ensure that different tasks can access the file system concurrently, you need to implement a few operating system-dependent functions.

For µC/OS-II and MS Windows, you will find implementations of these functions in the file system's source code. This chapter provides descriptions of the functions required to fully support µC/FS in multithreaded environments. If you do not use an OS, or if you do not make file access from different tasks, you can implement these functions as empty routines.

You may also add date and time support functions for use by the FAT file system. The sample implementations provided with µC/FS use ANSI C standard functions to obtain the correct date and time.

7.1 OS layer API functions

To use µC/FS with an operating system set the define `FS_OS` to `1` in `FS_Conf.h`. and implement the following functions. Samples for the implementation of an operating system can be found in the directory `\Sample\OS\`.

Example

7.2 Sample configuration

```
/*
*********************************************************************
*                           Micrium, Inc.
*                       949 Crestview Circle
*                      Weston,  FL 33327-1848
*
*                             uC/FS
*
*              (c) Copyright 2001 - 2006, Micrium, Inc.
*                       All rights reserved.
*
**********************************************************************
**** µC/FS file system for embedded applications ****
µC/FS is protected by international copyright laws. Knowledge of the
source code may not be used to write a similar product. This file may
only be used in accordance with a license and should not be re-
distributed in any way. We appreciate your understanding and fairness.
----------------------------------------------------------------------
File        : FS_Conf.h
Purpose     : File system configuration
---------------------------END-OF-HEADER------------------------------
*/

#ifndef _FS_CONF_H_
#define _FS_CONF_H_

#define FS_OS        1
#endif  /* Avoid multiple inclusion */
```

7.2.1 FS_X_OS_Init()

Description

Initializes the OS resources. Specifically, you will need to create at least NumLocks binary semaphores.

Prototype

```
void FS_X_OS_Init (unsigned NumLocks);
```

Parameter	Meaning
NumLocks	Number of binary semaphores/mutexes that should be created.

Table 7.1: FS_X_OS_Init() parameter list

Additional Information

This function is called by `FS_Init()`. You should create all resources required by the OS to support multithreading of the file system.

7.2.2 FS_X_OS_Lock()

Description

Locks a specific file system operation.

Prototype

```
void FS_X_OS_LockFileHandle (unsigned LockIndex);
```

Parameter	Meaning
`LockIndex`	Index number of the binary semaphore/mutex created before in `FS_X_OS_Init()`.

Table 7.2: FS_X_OS_Lock() parameter list

Additional Information

This routine is called by the file system before it accesses the device or before using a critical internal data structure. It blocks other threads from entering the same critical section using a resource semaphore/mutex until `FS_X_Unlock()` has been called with the same `LockIndex`.
When using a real time operating system, you normally have to increment a counting resource semaphore.

7.2.3 FS_X_OS_Unlock()

Description

Unlock FAT memory block table.

Prototype

```
void FS_X_OS_Unlock (unsigned LockIndex);
```

Parameter	Meaning
`LockIndex`	Index number of the binary semaphore/mutex created before in `FS_X_OS_Init()`.

Table 7.3: FS_X_OS_Unlock() parameter list

Additional Information

This routine is called by the file system after accessing the device or after using a critical internal data structure. When using a real time operating system, you normally have to decrement a counting resource semaphore.

7.2.4 Examples

OS interface routines for uC/OS

The following example shows an adaption for µC/OS (excerpt from file `FS_X_uCOS_II.c` located in the folder `\Sample\OS\`):

```
static  OS_EVENT  * _papSema;

void  FS_X_OS_Init (unsigned NumLocks) {
  unsigned i;
  OS_EVENT  * pSema;

  _papSema = (OS_EVENT  *)FS_AllocZeroed(NumLocks * sizeof(OS_EVENT));
  pSema    = _papSema;
  for(i = 0; i < NumLocks; i++) {
    pSema = OSSemCreate(1);
    pSema++;
  }
}

void  FS_X_OS_Unlock (unsigned LockIndex) {
  OS_EVENT  * pSema;

  pSema    = _papSema + LockIndex;
  OSSemPost(pSema);
}

void  FS_X_OS_Lock (unsigned LockIndex) {
  INT8U  err;
  OS_EVENT  * pSema;

  pSema    = _papSema + LockIndex;
  OSSemPend(pSema, 0, &err);
}
```

Chapter 8

Debugging

For debug purpose the following functions are helpful displaying information on display or through a serial communication port.

Furthermore, additional hints are provided which may be helpful if you run into trouble with µC/FS integration.

8.1 FS_X_Log()

Description

Outputs debug information from µC/FS. This function has to integrated into your application if `FS_DEBUG_LEVEL >= 5`. Refer to section "Debugging" on page 275 of the Configuration chapter for further information about the different debug-level.

Prototype

```
void FS_X_Log (const char * s);
```

Parameter	Meaning
`s`	Pointer to the string to be sent.

Table 8.1: FS_X_Log() parameter list

Example

```
/* sample using ANSI C printf function */

U16 FS_X_Log(const char* s) {
  printf("%s", s);
}
```

8.2 FS_X_Warn()

Description

Outputs warnings from µC/FS. This function has to integrated into your application if `FS_DEBUG_LEVEL >= 4`. Refer to section "Debugging" on page 275 of the Configuration chapter for further information about the different debug-level.

Prototype

```
void FS_X_Warn (const char * s);
```

Parameter	Meaning
s	Pointer to the string to be sent.

Table 8.2: FS_X_Warn() parameter list

Example

```
/* sample using ANSI C printf function */

U16 FS_X_Warn(const char* s) {
  printf("%s", s);
}
```

8.3 FS_X_ErrorOut()

Description

Outputs errors from µC/FS. This function has to integrated into your application if `FS_DEBUG_LEVEL >= 3`. Refer to section "Debugging" on page 275 of the Configuration chapter for further information about the different debug-level.

Prototype

```
void FS_X_ErrorOut (const char * s);
```

Parameter	Meaning
s	Pointer to the string to be sent.

Table 8.3: FS_X_ErrorOut() parameter list

Example

```
/* sample using ANSI C printf function */

U16 FS_X_ErrorOut(const char* s) {
  printf("%s", s);
}
```

8.4 Troubleshooting

If you are used to C-like file operations, you already know the `fopen()` function. In µC/FS, there is an equivalent function called `FS_FOpen()`. You specify a name, an access mode and if this kind of file access is allowed and no error occurs, you get a pointer to a file handle in return. For more information about the parameters refer to “FS_FOpen()” on page 48: Open a file

```
FS_FILE * pfile;
pfile = FS_FOpen("test.txt","r");
if (pFile == 0) {
  return -1; /* report error */
} else {
  return 0; /* file system is up and running! */
}
```

If this pointer is zero after calling `FS_FOpen()`, there was a problem opening the file. There are basically some common reasons why this could happen:

- The file or path does not exist
- The drive could not be read or written
- The drive contains an invalid BIOS parameter block or partition table

These faults can be caused by corrupted media. To verify the validity of your medium, either check if the medium is physically okay or check the medium with another operation system (for example Windows).

But there are also faults that are relatively seldom but also possible:

- A compiler/linker error has occurred
- Stack overflow
- Memory failure
- Electro-magnetic influence (EMC, EMV, ESD)

To find out what the real reason for the error is, you may just try reading and writing a raw sector. Here is an example function that tries writing a single sector to your device. After reading back and verifying the sector data, you know if sectored access to the device is possible and if your device is working.

```
int WriteSector(void) {
  U8  acBufferOut[FS_MAX_SECTOR_SIZE];
  U8  acBufferIn[FS_MAX_SECTOR_SIZE];
  U32 SecNum;
  int    x, i;
  SecNum = 80; /* Do not write on the first sectors. They contain
                  information about partitioning and media geometry. */
  for (i = 0; i < FS_MAX_SECTOR_SIZE; i++) { /* we fill the buffer with data */
    acBufferOut[i] = i % 256;
  }
  x = FS_IoCtl("",FS_CMD_WRITE_SECTOR, SecNum, acBufferOut);  /* Write one sector */
  if (x != 0) {
    FS_X_Log("Cannot write to sector.\n");
    return -1;
  }
  x = FS_IoCtl("",FS_CMD_READ_SECTOR, SecNum, acBufferIn); /* read the sector */
  if (x != 0) {
    FS_X_Log("Cannot read from sector.\n");
    return -1;
  }
  for (i = 0; i < FS_MAX_SECTOR_SIZE; i++) {
    if (acBufferIn[i] != acBufferOut[i]) {
      FS_X_Log("Sector not correctly written.\n");
      return -1;
    }
  }
  return 0;
}
```

If you still receive no valid file pointer although the sectors of the device is accessible and other operating systems report the device to be valid, you may have to take a look into the running system by stepping through the function `FS_FOpen()`.

Chapter 9

Performance and Resource usage

9.1 Memory footprint

The file system is designed to fit many kinds of embedded design requirements. Several features can be excluded from a build to get a minimal system that can efficiently access any FAT media.

The operation area of µC/FS is very different and the memory requirements (RAM and ROM) differs in depending on the used features. The following section will show the memory requirements of different modules which are used in typical applications.

Note that the values are valid for the given configuration. Features can affect the size of others. For example, if FAT32 is deactivated, the format function gets smaller because the 32 bit specific part of format is not added into the compilation.

9.1.1 System

The following table shows the hardware and the toolchain details of the project:

Detail	Description
CPU	ARM7
Tool chain	IAR Embedded Workbench for ARM V4.41A
Model	ARM7, Thumb instructions; no interwork;
Compiler options	Highest size optimization;
Device driver	Empty dummy driver. For information about the memory usage of a specific µC/FS device driver refer to the Unit number section of the respective driver in the "Device drivers" on page 141.

Table 9.1: Arm7 sample configuration

9.1.2 File system configuration

The following excerpts of `FS_Conf.h` shows the used configuration options:

```
#define FS_OS_LOCKING                0 // Disable OS support
#define FS_SUPPORT_FAT               1 // Support the FAT file system if enabled
#define FS_DEBUG_LEVEL               0 // Set debug level
```

9.1.3 Sample project

We use the following code to calculate the memory resources of commonly used functions. You can easily reproduce the measurement when you compile the following sample. Build the application listed below and generate a linker listing to get the memory requirements of an application which only includes startup code and the-empty `main()` function. Afterwards, set the value of the macro `STEP` to 1 to get the memory requirement of the minimum file system. Subtract the ROM requirements from `STEP==0` from the ROM requirements of `STEP==1` to get the exact ROM requirements of a minimal file system. Increment the value of the macro STEP to include more file system functions and repeat your calculation.

```
#include "FS.h"'
#include "FS_Int.h"

/*********************************************************************
*
*       defines, configurable
```

```
*
**********************************************************************/
#define STEP    0     // Change this line to adjust which portions of code are linked

/*********************************************************************
*
*       Public code
*
**********************************************************************/

/*********************************************************************
*
*       main
*/
void main(void) {
#if STEP >= 1            // Step 1: Minimum file system
  FS_FILE    * pFile;
  FS_Init();
  pFile = FS_FOpen("File.txt", "w");
  FS_FClose(pFile);
#endif
#if STEP >= 2           // Step 2: Write a file
  FS_Write(pFile, "Test", 4);
#endif
#if STEP >= 3           // Step 3: Remove a file
  FS_Remove("File.txt");
#endif
#if STEP >= 4           // Step 4: Open a directory
  FS_FIND_DATA fd;
  FS_FindFirstFile(&fd, "\\YourDir\\", "File.txt", 8);
  FS_FindClose(&fd);
#endif
#if STEP >= 5          // Step 5: Create a directory
  FS_MkDir ("");
#endif
#if STEP >= 6          // Step 6: Add long file name support
  FS_FAT_SupportLFN();
#endif
#if STEP >= 7          // Step 7: Low-level format a medium
  FS_FormatLow("");
#endif
#if STEP >= 8          // Step 8: High-level format a medium
  FS_Format("", NULL);
#endif
#if STEP >= 9          // Step 9: Assign cache - Cache module: FS_CACHE_ALL
  FS_AssignCache("", NULL, 0, FS_CACHE_ALL);
//  FS_AssignCache("", NULL, 0, FS_CACHE_MAN);
//  FS_AssignCache("", NULL, 0, FS_CACHE_RW);
//  FS_AssignCache("", NULL, 0, FS_CACHE_RW_QUOTA);
#endif
#if STEP >= 10         // Step 10: Checkdisk
  FS_FAT_CheckDisk("", NULL, 0, 0, NULL);
#endif
#if STEP >= 11         // Step 11: Get device info
  FS_GetDeviceInfo("", NULL);
#endif
#if STEP >= 12         // Step 12: Get the size of a file
  FS_GetFileSize(NULL);
#endif
}
```

9.1.4 Memory requirements

The following table shows the memory requirement of the used functions:

Description	ROM [Kbytes]
Step 1: Minimum file system (without any driver)	7.0
Step 2: Write a file	1.1
Step 3: Remove a file	0.1
Step 4: Open directory	0.5
Step 5: Create directory	0.5
Step 6: Long file name support	2.0
Step 7: Low-level format a medium	0.2
Step 8: High-level format a medium	1.8
Step 9: Assign a cache - `FS_CACHE_ALL`	0.4
Assign a cache - `FS_CACHE_MAN`	0.7
Assign a cache - `FS_CACHE_RW`	0.7
Assign a cache - `FS_CACHE_RW_QUOTA`	1.0
Step 10: Checkdisk	3.3
Step 11: Get device info	0.1
Step 12: Get the size of a file	0.1

The static RAM requirement of the file system without any driver is around 2.5 Kbytes.

9.2 Performance

A benchmark is used to measure the speed of the software on available targets. This benchmark is in no way complete, but it gives an approximation of the length of time required for common operations on various targets.

9.2.1 Configuration and performance table

All values are in Kbytes/sec.

System	Medium	W0	W1	R
ARM 7 (50 MHz)	Hard drive	2431	2434	4167
ARM 7 (50 MHz)	Compact Flash, IDE mode	1882	1882	2000
ARM 7 (50 MHz)	Compact Flash, mem-mapped	2782	2782	2560
ARM 7 (50 MHz)	MMC / SD using SPI with 12MHz	1280	1333	1488
XScale (208 MHz)	MMC / SD using SD4 card mode with 20MHz	7262	7262	5389
ARM 7 (50 MHz)	SMC	392	390	771
ARM 7 (50 MHz)	NAND Flash 8MB	474	474	771
ARM 7 (33 MHz)	NOR flash	172	91	1254
ARM 7 (50 MHz)	RAM disk	4000	4000	4571

Table 9.2: Performance valus for sample configurations

9.2.1.1 Description of the performance tests

The performance tests are executed as described and in the order below.
Performance test procedure:

1. Format the drive.
   W0: Writing
2. Open a file for writing .
3. Write data to the opened file in chunks of 64 kBytes, using 80% of the volume capacity but no more than 4 MByte. Writing a 64KByte chunk is measured.
4. Close the file
   W1: Writing
5. Reopen the file for writing (old created file will be deleted).
6. Write data to the opened file in chunks of 64kBytes, using 80% of the capacity of the medium but no more than 4MByte. Writing a 64KByte chunk is measured.
7. Close the file.
   R: Reading
8. Open the file for reading
9. Read 4 MBytes from file in 64KByte chunks. Reading a 64KByte chunk is measured. (R1)
10. Close the file.
11. Show the performance results.

Chapter 10

Porting µC/FS 2.x to 3.x

10.1 Differences from version 2.x to 3.x

Most of the differences from µC/FS version 2.x to version 3.x are internal. The API of µC/FS version 2.x is a subset of the API of version 3.x. Only few functions are completely removed. Refer to section "API differences" on page 297 for a complete overview of the removed and obsolete functions.

µC/FS version 3 has a new driver handling. You can include drivers and allocate the required memory for the accordant driver without the need to recompile the whole file system. Refer to "µC/FS Configuration" on page 267 for detailed information about the integration of a driver into µC/FS. For detailed information to the µC/FS device drivers, refer to the chapter "Device drivers" on page 141.

Because of these differences, we recommend to start with a new file system project and include your application code, if the start project runs without any problem. Refer to the chapter "Running µC/FS on target hardware" on page 21 for detailed information about the best way to start the work with µC/FS version 3.x.

The following sections gives an overview about the changes from µC/FS version 2.x. to µC/FS version 3 in table form.

10.2 API differences

Function	Description
Changed functions	
`FS_GetFreeSpace()`	Number of parameters reduced. Parameter `DevIndex` removed.
`FS_GetTotalSpace()`	Number of parameters reduced. Parameter `DevIndex` removed.
Removed functions	
`FS_Exit()`	Should be removed from your application source code.
`FS_CheckMediumPresent()`	
Obsolete directory handling functions	
`FS_CloseDir()`	The directory handling has been changed in µC/FS version 3.x. The functions should be replaced. Refer to "FS_FindClose()" on page 73 for an example of the new way of directory handling.
`FS_DirEnt2Attr()`	
`FS_DirEnt2Name()`	
`FS_DirEnt2Size()`	
`FS_DirEnt2Time()`	
`FS_GetNumFiles()`	
`FS_OpenDir()`	
`FS_ReadDir()`	
`FS_RewindDir()`	
Obsolete file system extended functions	
`FS_IoCtl()`	`FS_IoCtl()` should not be used in µC/FS version 3.x. Use `FS_IsLLFormatted()` to check if a low-level format is required and `FS_GetDeviceInfo()` to get the device information.

Table 10.1: Differences between µC/FS v.2.x / v.3.x - API differences

In µC/FS version 3 is the header file `FS_Api.h` renamed to `FS.h`, therefore change the name of the file system header file in your application.

10.3 Configuration differences

The configuration of μC/FS version 3.x has been simplified compared to μC/FS v2.x. μC/FS v3.x can be used "out of the box". You can use it without the need for changing any of the compile time flags. All compile time configuration flags are preconfigured with valid values, which matchs the requirements of the most applications.

A lot of the compile time flags of μC/FS v.2.x are removed and replaced with runtime configuration function.

Removed/replaced configuration macros

In version 3.x removed macros	In version 3.x used macros
File system configuration	
`FS_MAXOPEN`	`FS_NUM_FILE_HANDLES`
`FS_POSIX_DIR_SUPPORT`	--
`FS_DIR_MAXOPEN`	`FS_NUM_DIR_HANDLES`
`FS_DIRNAME_MAX`	--
`FS_SUPPORT_BURST`	--
`FS_DRIVER_ALIGNMENT`	--
FAT configuration macros	
`FS_FAT_SUPPORT_LFN`	Replaced by `FS_FAT_SupportLFN()`. Refer to "FS_FAT_SupportLFN()" on page 115 for more information.

Table 10.2: Differences between μC/FS v.2.x / v.3.x - removed/replaced configuration macros

Changed default values of configuration macros

Changed macros	Default value in
`FS_FAT_OPTIMIZE_SEQ_CLUSTERS`	Default value changed from 0 to 1.
`FS_EFS_FWRITE_UPDATE_DIR`	Default value changed from 1 to 0.

Table 10.3: Differences between μC/FS v.2.x / v.3.x - changed default values

10.4 Device driver

10.4.1 Renamed drivers

Old driver names	Driver names in µC/FS version 3.x
NAND2K	In µC/FS version 3.x, the NAND driver could be used to access small and large block NAND flashes similarly. The driver is therefore renamed from NAND2K to NAND.
SMC	In µC/FS version 3, the SMC / small block NAND driver is integrated in the NAND driver. The NAND driver could be used to access small and large block NAND flashes similarly.
SFLASH	The serial flash driver is renamed into DataFlash driver.
FLASH	FLASH driver renamed to NOR flash driver.

Table 10.4: Differences between µC/FS v.2.x / v.3.x - list of renamed device drivers

10.4.2 Integrating a device driver into µC/FS

In version 2.x, you have to enable a device driver with a macro which has to be set has to be set in the µC/FS configuration file `FS_Conf.h` and recompile your file system project. µC/FS version 3.x is run time configurable, so that you can add all device drivers by calling the `FS_AddDevice()` function with the proper parameter for the accordant driver.

In version 3.x removed macros	Alternative
FS_USE_FLASH_DRIVER	FS_AddDevice(&FS_NOR_Driver)
FS_USE_IDE_DRIVER	FS_AddDevice(&FS_IDE_Driver)

FS_USE_MMC_DRIVER	FS_AddDevice(&FS_MMC_SPI_Driver)
	FS_AddDevice(&FS_MMC_CardMode_Driver)

FS_USE_RAMDISK_DRIVER	FS_AddDevice(&FS_RAMDISK_Driver)
FS_USE_SFLASH_DRIVER	FS_AddDevice(&FS_DataFlash_Driver)
FS_USE_SMC_DRIVER	FS_AddDevice(&FS_NAND_Driver)
FS_USE_NAND2K_DRIVER	FS_AddDevice(&FS_NAND_Driver)
FS_USE_WINDRIVE_DRIVER	FS_AddDevice(&FS_WINDRIVE_Driver)

Table 10.5: Differences between µC/FS v.2.x / v.3.x - adding a driver

10.4.3 RAM disk driver differences

In version 3.x removed macros	Alternative
FS_USE_RAMDISK_DRIVER	FS_AddDevice(&FS_RAMDISK_Driver)
FS_RAMDISK_NUM_SECTORS	FS_RAMDISK_Configure() - Refer to "FS_RAMDISK_Configure()" on page 144 for detailed information.
FS_RAMDISK_MAXUNIT	
FS_RAMDISK_ADDR	
FS_RAMDISK_SECTOR_SIZE	

Table 10.6: Differences between µC/FS v.2.x / v.3.x - removed RAMDISK macros

Refer to the section "RAM disk driver" on page 143 for detailed information about the RAM disk driver in µC/FS version 3.x.

10.4.4 NAND driver differences

In version 3.x removed macros	Alternative
FS_USE_NAND2K_DRIVER	FS_AddDevice(&FS_NAND_Driver)
FS_NAND2K_MAXUNIT	FS_NAND_SetPhyType() - Refer to "FS_NAND_SetPhyType()" on page 155 for detailed information.
FS_NAND2K_MAX_NUM_PHY_BLOCKS	FS_NAND_SetBlockRange() - Refer to "FS_NAND_SetBlockRange()" on page 156 for detailed information.

Table 10.7: Differences between µC/FS v.2.x / v.3.x - removed NAND driver macros

Hardware interface version 2.x	Hardware interface version 3.x
FS_NAND2K_HW_X_SetAddr()	FS_NAND_HW_X_SetAddrMode()
FS_NAND2K_HW_X_SetCmd()	FS_NAND_HW_X_SetCmdMode()
FS_NAND2K_HW_X_SetData()	FS_NAND_HW_X_SetDataMode()
FS_NAND2K_HW_X_SetStandby()	FS_NAND_HW_X_SetStandby()
FS_NAND2K_HW_X_WaitWhileBusy()	FS_NAND_HW_X_WaitWhileBusy()
FS_NAND2K_HW_X_IsWriteProtected()	FS_NAND_HW_X_IsWriteProtected()
FS_NAND2K_HW_X_Read()	FS_NAND_HW_X_Read()
FS_NAND2K_HW_X_Write()	FS_NAND_HW_X_Write()
FS_NAND2k_HW_X_Delayus()	FS_NAND_HW_X_Delayus()
FS_NAND2K_HW_X_Init()	FS_NAND_HW_X_Init()
--	FS_NAND_HW_X_DisableCE()
--	FS_NAND_HW_X_EnableCE()

Table 10.8: Differences between µC/FS v.2.x / v.3.x - IDE driver hardware interface differences

Refer to the section "NAND flash driver" on page 147 for detailed information about the NAND driver in µC/FS version 3.x.

10.4.5 NAND driver differences

In version 3.x removed macros	Alternative
FS_USE_SMC_DRIVER	FS_AddDevice(&FS_NAND_Driver)
FS_SMC_MAXUNIT	FS_NAND_SetPhyType() - Refer to "FS_NAND_SetPhyType()" on page 155 for detailed information. FS_NAND_SetBlockRange() - Refer to "FS_NAND_SetBlockRange()" on page 156 for detailed information.
FS_SMC_HW_SUPPORT_BSYLINE_CHECK	

Table 10.9: Differences between µC/FS v.2.x / v.3.x - adding a driver

Hardware interface version 2.x	Hardware interface version 3.x
FS_SMC_HW_X_SetAddr()	In µC/FS version 3, the SMC / small block NAND driver is integrated in the NAND driver. The NAND driver could be used to access small and large block NAND flashes similarly. Refer to "NAND flash driver" on page 147 for detailed inforamtion about the NAND driver in µC/FS version 3.x
FS_SMC_HW_X_SetCmd()	
FS_SMC_HW_X_SetData()	
FS_SMC_HW_X_SetStandby()	
FS_SMC_HW_X_VccOff()	
FS_SMC_HW_X_VccOn()	
FS_SMC_HW_X_ChkBusy()	
FS_SMC_HW_X_ChkCardIn()	
FS_SMC_HW_X_ChkPower()	
FS_SMC_HW_X_ChkStatus()	
FS_SMC_HW_X_ChkWP()	
FS_SMC_HW_X_DetectStatus()	
FS_SMC_HW_X_InData()	
FS_SMC_HW_X_OutData()	
FS_SMC_HW_X_ChkTimer()	
FS_SMC_HW_X_SetTimer()	
FS_SMC_HW_X_StopTimer()	
FS_SMC_HW_X_WaitTimer()	

Table 10.10: Differences between µC/FS v.2.x / v.3.x - IDE driver hardware interface differences

Refer to the section "NAND flash driver" on page 147 for detailed information about the NAND driver in µC/FS version 3.x.

10.4.6 MMC driver differences

In version 3.x removed macros	Alternative
FS_USE_MMC_DRIVER	FS_AddDevice(&FS_MMC_CardMode_Driver)
FS_MMC_USE_SPI_MODE	FS_AddDevice(&FS_MMC_SPI_Driver)
FS_MMC_MAXUNIT	--
FS_USE_CRC	FS_MMC_ActivateCRC() / FS_MMC_DeactivateCRC()
FS_MMC_SUPPORT_4BIT_MODE	--

Table 10.11: Differences between µC/FS v.2.x / v.3.x - removed MMC macros

Refer to the section “MultiMedia & SD card driver” on page 188 for detailed information about the MMC driver in µC/FS version 3.x.

10.4.7 CF/IDE driver differences

In version 3.x removed macros	Alternative
FS_USE_IDE_DRIVER	FS_AddDevice(&FS_IDE_Driver)
FS_IDE_MAXUNIT	--

Table 10.12: Differences between µC/FS v.2.x / v.3.x - removed CF/IDE macros

In version 3.x is the hardware inteface of the CF/IDE driver simplified. Only 6 hardware functions have to implemented.

Hardware interface version 2.x	Hardware interface version 3.x
FS_IDE_HW_X_HWReset()	FS_IDE_HW_X_HWReset()
FS_IDE_HW_X_Delay400ns()	FS_IDE_HW_X_Delay400ns()
FS_IDE_HW_X_GetAltStatus()	--
FS_IDE_HW_X_GetCylHigh()	--
FS_IDE_HW_X_GetCylLow()	--
FS_IDE_HW_X_GetData()	FS_IDE_HW_X_ReadData()
FS_IDE_HW_X_GetError()	--
FS_IDE_HW_X_GetSectorCount()	--
FS_IDE_HW_X_GetSectorNo()	--
FS_IDE_HW_X_GetStatus()	--
FS_IDE_HW_X_SetCommand()	--
FS_IDE_HW_X_SetCylHigh()	--
FS_IDE_HW_X_SetCylLow()	--
FS_IDE_HW_X_SetData()	FS_IDE_HW_X_WriteData()
FS_IDE_HW_X_SetDevControl()	--
FS_IDE_HW_X_SetDevice()	--
FS_IDE_HW_X_SetFeatures()	--
FS_IDE_HW_X_SetSectorCount()	--
FS_IDE_HW_X_SetSectorNo()	--

Table 10.13: Differences between µC/FS v.2.x / v.3.x - CF/IDE driver hardware interface differences

Hardware interface version 2.x	Hardware interface version 3.x
`FS_IDE_HW_X_DetectStatus()`	--
--	`FS_IDE_HW_X_ReadReg()`
--	`FS_IDE_HW_X_WriteReg()`

Table 10.13: Differences between µC/FS v.2.x / v.3.x - CF/IDE driver hardware interface differences

Refer to the section "CompactFlash card & IDE driver" on page 223 for detailed information about the CF/IDE driver in µC/FS version 3.x.

10.4.8 Flash / NOR flash differences

In version 3.x removed macros	Alternative
`FS_USE_FLASH_DRIVER`	`FS_AddDevice(&FS_NOR_Driver)`
`FS_FLASH_MAX_ERASE_CNT_DIFF`	`FS_NOR_Configure()` - Refer to "FS_NOR_Configure()" on page 248 for detailed information. `FS_NOR_SetPhyType()` - Refer to "FS_NOR_SetPhyType()" on page 249 for detailed information.
`FS_FLASH_NUM_FREE_SECTORCACHE`	
`FS_FLASH_CHECK_INFO_SECTOR`	
`FLASH_BASEADR`	
`FLASH_USER_START`	
`FLASH_BYTEMODE`	
`FLASH_RELOCATECODE`	
`FS_FLASH_CAN_REWRITE`	
`FS_FLASH_LINE_SIZE`	
`FS_FLASH_SECTOR_SIZE`	

Table 10.14: Differences between µC/FS v.2.x / v.3.x - removed Flash / NOR flash macros

Refer to the section "NOR flash driver" on page 244 for detailed information about the NOR flash driver in µC/FS version 3.x.

10.4.9 Serial Flash / DataFlash differences

In version 3.x removed macros	Alternative
`FS_USE_SFLASH_DRIVER`	`FS_AddDevice(&FS_DataFlash_Driver);`
`FS_SFLASH_MAXUNIT`	--

Table 10.15: Differences between µC/FS v.2.x / v.3.x - removed Serial Flash / DataFlash macros

Note: Since version 3.10 is the DataFlash support integrated in the NAND flash driver. Refer to "NAND flash driver" on page 147 for detailed information.

10.4.10 Windrive differences

In version 3.x removed macros	Alternative
FS_WD_DEV0NAME	FS_Windrive_Configure() - Refer to “FS_Windrive_Configure()” on page 255 for detailed information.
FS_WD_DEV1NAME	

Table 10.16: Differences between µC/FS v.2.x / v.3.x - removed Windrive macros

Refer to the section “WinDrive driver” on page 255 for detailed information about the Windrive driver in µC/FS version 3.x.

10.5 OS Integration

In version 3.x removed macros	In version 3.x used macros
OS configuration macros	
FS_OS_LOCKING_PER_FILE	Removed. If you want to use µC/FS version 3.x with an RTOS, define FS_OS_LOCKING in your FS_Conf.h. Refer to “OS integration” on page 271 for information about he functions which has to be implemented to use µC/FS with an RTOS.
FS_OS_EMBOS	
FS_OS_UCOS_II	
FS_OS_WINDOWS	
FS_OS	

Table 10.17: Differences between µC/FS v.2.x / v.3.x - removed/replaced configuration macros

Function	Description
Changed functions	
FS_X_OS_Init()	In µC/FS version 3.x gets FS_X_OS_Init() an additional parameter. Refer to FS_X_OS_Init()
Removed functions	
FS_X_OS_ Exit()	--
Time and date functions	
FS_X_OS_GetDate()	In µC/FS version 3.x is only one version used to handle the time and date functionality. Refer to “FS_X_GetTimeDate()” on page 264 for more information.
FS_X_OS_GetDateTime()	

Table 10.18: Differences between µC/FS v.2.x / v.3.x - Changes in the OS interface

Chapter 11

FAQs

You can find in this chapter a collection of frequently asked questions (FAQs) together with answers.

11.1 FAQs

Q: Is my data safe, when an unexpected RESET occurs?

A: In general, the data which is already on the medium is safe. If a read operation is interrupted, this is completely harmless. If a write operation is interrupted, the data written in this operation may or may not be stored on the medium, depending on when the unexpected RESET occurred. In any case, the data which was on the media prior to the write operation is not affected; directory entries are not messed up, the file-allocation-table is kept in order. This is true if your storage medium is not affected by the RESET, meaning that it is able to complete a pending write operation. (Which is typically the case with Flash memory cards other than SMC)

Q: I use FAT and I can only create a limited number of root directory entries. Why?

A: With FAT12 and FAT16 the root directory is special because it has a fixed size. During media format one can determine the size, but once formatted this value is constant and determines the number of entries the root directory can hold. FAT32 does not have this limitation and the root directory's size can be variable.

Microsoft's "FAT32 File System Specification" says on page 22: "For FAT12 and FAT16 media, the root directory is located in a fixed location on the disk immediately following the last FAT and is of a fixed size in sectors computed from the BPB_RootEntCnt value [...] For FAT32, the root directory can be of variable size and is a cluster chain, just like any other directory is.". Here BPB_RootEntCnt specifies the count of 32-byte directory entries in the root directory and as the citation says, the number of sectors is computed from this value.

In addition, which file system is used depends on the size of the medium, that is the number of clusters and the cluster size, where each cluster contains one or more sectors. Using small cluster sizes (for example cluster size = 512 bytes) one can use FAT32 on media with more than 32 MB. (FAT16 can address at least 216 clusters with a 512 byte cluster size. That is 65536 * 512 = 33554432 bytes = 32768 KB = 32 MB). If the media is smaller than or equal to 32 MB or the cluster size is greater than 512 bytes, FAT32 cannot be used.

To actually set a custom root directory size for FAT12/FAT16 one can use the µC/FS API function `int FS_Format(const char *pDevice, FS_FORMAT_INFO * pFormatInfo);` where `FS_FORMAT_INFO` is declared as:

```
typedef struct {
  U16 SectorsPerCluster;
  U16 NumRootDirEntries;
  FS_DEV_INFO * pDevInfo;
} FS_FORMAT_INFO;
```

Set `NumRootDirEntries` to the desired number of root directory entries you want to store.

Index